studien—texte Informatik

Herausgegeben von Jürgen Perl

studien—texte

Informatik / Herausgegeben von Jürgen Perl

Günter Hommel / Joachim Jäckel / Stefan Jähnichen
Karl Kleine / Wilfried Koch / Kees Koster
ELAN-Sprachbeschreibung
1979, 108 Seiten, kart., DM 14,80
ISBN 3-400-00384-0

Rainer Hahn / Peter Stock
ELAN-Handbuch
1979, 160 Seiten, kart., DM 19,80
ISBN 3-400-00386-7

In Vorbereitung:

Jürgen Ebert
Graphenalgorithmen

Karl Heinz Sturm
Mikroprozessoren

Peter Brucker
Scheduling

Rainer Hahn
Peter Stock

ELAN-Handbuch

Akademische Verlagsgesellschaft
Wiesbaden 1979

Dipl.-Ing. R. Hahn
H. Peter Stock
Hochschul-Rechenzentrum
Universität Bielefeld
Postfach 8640
4800 Bielefeld 1

10
HI 450
E 270
H148

Univ.
Bibliothek
Bielefeld

CIP-Kurztitelaufnahme der Deutschen Bibliothek

Hahn, Rainer:
ELAN-Handbuch / Rainer Hahn; Peter Stock. –
Wiesbaden: Akademische Verlagsgesellschaft, 1979.

(Studien-Texte: Informatik)
ISBN 3-400-00386-7

83-84/841

BI108/1223397+01

© 1979 by Akademische Verlagsgesellschaft, Wiesbaden
Ohne ausdrückliche Genehmigung des Verlages ist es auch nicht gestattet, dieses Buch oder Teile daraus auf photomechanischem Wege (Photokopie, Mikrokopie) zu vervielfältigen.
Printed in Germany/Imprimé en Allemagne
Gesamtherstellung: Decker + Wilhelm, Heusenstamm
ISSN 0172-2239
ISBN 3-400-00386-7

INHALTSVERZEICHNIS

Seite

Vorwort

1. Die Programmiersprache ELAN..... 6

1.1. Übersicht über die Programmiersprache ELAN 6
1.1.1. Was nicht vorhanden ist 6
1.1.2. Kurze Übersicht 6
1.2. Überblick über die Entwicklung der Programmiersprache ELAN 11
1.3. Entwurfskriterien der Programmiersprache ELAN 13

2. Die lexikalischen Elemente..... 14

2.1. Schlüsselwörter 14
2.2. Bezeichner (Namen) 15
2.3. Sonderzeichen 15
2.4. Kommentare 16

3. Die Verwendung einfacher Datentypen..... 17

3.1. Datentypen und Datenobjekte in ELAN 17
3.2. Die einfachen Datentypen 18
3.3. Denotierungen der einfachen Datentypen 18
3.4. Das Accessrecht 19
3.5. Die Vereinbarung von Datenobjekten (Deklaration) 20

4. Operatoren für einfache Datentypen..... 22

4.1. Objekte und Elaboration 22
4.2. Monadische und dyadische Operatoren 22
4.2.1. Einige monadische Operatoren 24
4.2.2. Einige dyadische Operatoren 25
4.3. Ausdrücke (Formeln) und Prioritäten 26
4.4. Die Zuweisung 27
4.5. Deklaration mit Initialisierung 28
4.6. Wandlung des Accessrechts 29
4.6.1. "consting" 29
4.6.2. Balancierungen des Accessrechts 30

5. Zusammengesetzte Daten..... 31

5.1. Die Zusammenfassung gleichartiger Datentypen 31
5.1.1. Zugriff auf die Komponenten einer Reihung (Subskription) 32
5.1.2. Reihungs-Denoter ("row display") 32
5.1.3. Reihungen von Reihungen 33
5.1.4. Subskription und "row display" von mehrfachen Reihungen 34
5.2. Die Zusammenfassung ungleichartiger Datentypen 34
5.2.1. Zugriff auf Komponenten von Strukturen (Selektion) 35
5.2.2. Denoter für Strukturtypen (Konstruktor) 35
5.2.3. Strukturen von Strukturen 36

Seite

5.3. Die Kombination von Reihungen und Strukturen 36
5.3.1. Reihungen innerhalb von Strukturen 36
5.3.2. Reihungen von Strukturen 37
5.3.3. Reihungen von Strukturen, die Reihungen enthalten 38

6. Zusammengesetzte Operationen..... 39

6.1. Einheiten ("units") 39
6.2. Die Aneinanderreihung von Einheiten (Sequenz) 39
6.3. Abfragen ("choice clause") 40
6.3.1. Die einseitige Abfrage 40
6.3.2. Die zweiseitige Abfrage 41
6.3.3. Die Abfragekette 42
6.4. Die Auswahl ("multiple choise clause") 43
6.5. Wiederholungen ("repetitions") 44
6.5.1. Die Endlosschleife 44
6.5.2. Die abweisende Schleife (WHILE-Konstrukt) 44
6.5.3. Die nichtabweisende Schleife (UNTIL-Konstrukt) 45
6.5.4. Die Zählschleife (FOR-Konstrukt) 46
6.6. Dynamische Initialisierung 47

7. Die schwache Datenabstraktion (LET-Konstrukt)..... 48

7.1. Namen für Denoter 48
7.2. Namen für Datentypen 49
7.3. Gemeinsame Verwendung von Namen für Denoter und Datentypen 49

8. Refinements: die schwache algorithmische Abstraktion..... 50

8.1. Das Refinement als Instrument zur schrittweisen Verfeinerung 50
8.2. Die Konstruktion von Refinements 54
8.3. Werte liefernde Einheiten 55
8.4. Werte liefernde Refinements 57
8.5. Terminatoren (LEAVE-Konstrukt) 59

9. Die starke Datenabstraktion..... 61

9.1. Die Definition neuer Datentypen 61
9.1.1. Denoter für abstrakte Datentypen (Konstruktor) 62
9.1.2. Zugriff auf die Feinstruktur (Konkretisierung) 62
9.2. Beispiele für Feinstrukturen neudefinierter Datentypen 63
9.2.2. Feinstruktur: Reihung 64
9.2.3. Feinstruktur: Struktur 64

10. Die starke algorithmische Abstraktion..... 65

10.1. Prozeduren 65
10.1.1. Prozeduren ohne Parameter 65
10.1.2. Prozeduren mit Parametern 67
10.1.2.1. Datenobjekte als Parameter 67
10.1.2.2. Prozeduren als Parameter 68

Seite

10.1.3. Generizität von Prozeduren 69
10.1.4. Werte liefernde Prozeduren 70
10.2. Operatoren 71
10.2.1. Neudefinierte monadische Operatoren 71
10.2.2. Neudefinierte dyadische Operatoren 72
10.2.3. Die Priorität neudefinierter Operatoren 72
10.3. Terminatoren für Prozeduren und Operatoren 74

11. Pakete..... 75

11.1. Module 75
11.1.1. Formulierung eines Pakets 76
11.1.2. Schnittstelle eines Pakets 77
11.1.3. Paketrumpf ("packet body") 78
11.2. Programmstruktur 78
11.2.1. Reihenfolge und Struktur von Paketen 78
11.2.2. Gültigkeitsbereich von Objekten 80
11.2.3. Überdeckungen 82
11.3. Abstraktionsebenen 83

12. Dateien ("files")..... 89

12.1. Dateiarten 89
12.2. Deklaration und Assoziierung von Dateien 90
12.3. Verwaltung von Dateien 91
12.4. Ein-/Ausgabe 92
12.4.1. Datenobjektorientierte Eingabe 92
12.4.2. Datenobjektorientierte Ausgabe 96
12.4.3. Zeilenorientierte Ein-/Ausgabe 97
12.4.4. Zeilenverwaltung 100
12.4.5. Informationsprozeduren 102

13. Die Standard-Pakete..... 103

13.1. Standard der einfachen Datentypen 103
13.1.1. Datentyp INT 104
13.1.2. Datentyp REAL 105
13.1.3. Datentyp BOOL 107
13.1.4. Datentyp TEXT 107
13.2. Standard der abstrakten Datentypen 112
13.2.1. Datentyp COMPLEX 112
13.3. Standard der mathematischen Routinen und Konstanten 116
13.4. Standard der Konvertierungen 118
13.4.1. Umwandlung in einen Text 118
13.4.2. Erkennen numerischer Werte 120
13.5. Besonderheiten 121

Anhang A: Schlüsselwörter mit einer festen Bedeutung ("bolds") 123
Anhang B: Kleines ELAN-Wörterbuch 124
Anhang C: ELAN-Syntaxdiagramme 130
Literatur 148
Stichwort-Verzeichnis 149

Vorwort

Dieses Handbuch beschreibt die Ausbildungssprache ELAN (Elementary LANguage). Es ist keine "Einführung in die Programmierung mit ELAN". Wir setzen daher grundlegende Kenntnisse der Programmierung als bekannt voraus. Der Leser sollte Begriffe wie Algorithmen, Datentyp, Zuweisung, Iteration und Rekursion, Prozeduren, Parameter usw. kennen. Trotzdem kann dieses Buch als Begleitmaterial für einen Einführungskurs in ELAN dienen, wenn die obengenannten Kenntnisse zusätzlich vermittelt werden.

Die Abfassung dieses Buches geschah unter folgenden Gesichtspunkten:

a) Das Buch soll einerseits als Lehrbuch dienen, d.h. es kann "sequentiell" gelesen werden. Andererseits soll es auch als Nachschlagewerk dienen, d.h. einzelne Abschnitte sollen - ohne daß die vorhergehenden Kapitel gelesen werden müssen - verständlich sein.

b) Die Darstellung des Stoffes erfolgt "bottom up", d.h. es werden zuerst einzelne und besondere Konstrukte sowie Beispiele dargestellt, die später in umfassendere einmünden.

c) Um den untrennbaren Zusammenhang zwischen den Daten und Algorithmen (Operationen) darzustellen, werden diese parallel in fünf Stufen entwickelt.

Diese fünf Stufen sind:

Einfache Daten - Kap. 3 -	Operatoren und Zuweisung - Kap. 4 -
Zusammengesetzte Daten (Reihungen und Strukturen) - Kap. 5 -	Einheiten und Kontrollstrukturen - Kap. 6 -
Schwache Datenabstraktion (Benennungen, Abkürzungen) - Kap. 7 -	Schwache algorithmische Abstraktion (Refinements) - Kap. 8 -
Starke Datenabstraktion (Neudefinition von Datentypen) - Kap. 9 -	Starke algorithmische Abstraktion (Prozeduren und neudefinierte Operatoren) - Kap. 10 -

Abstraktionsebenen
(Module, abstrakte Datentypen)
- Kap. 11 -

Dieses Handbuch ersetzt nicht die Sprachbeschreibung der Programmiersprache ELAN, die als gültige Definition in allen Zweifelsfällen herangezogen werden sollte. Zum Zeitpunkt der Drucklegung dieses Bandes gilt die Version 2.3 vom Frühjahr 1979.

Wir haben uns bemüht, in dieser ersten Version des Handbuches ein leicht verständliches und fehlerfreies Manuskript zu erstellen. Trotz aller Bemühungen können unklare Formulierungen oder sogar Fehler hierin enthalten sein. Für Korrekturen und Anmerkungen wären wir dankbar.

Wir danken allen Kollegen und Freunden, die uns mit ihren kritischen Anmerkungen geholfen haben.

1. Die Programmiersprache ELAN

Die Eigenschaften der Programmiersprache ELAN, die Entstehungsgeschichte sowie die Entwurfskriterien, die maßgeblich zur Herausbildung der Sprachelemente gedient haben, werden in diesem Kapitel dargestellt.

1.1. Übersicht über die Programmiersprache ELAN

Zunächst wollen wir in diesem Abschnitt einen kurzen Überblick über die programmiersprachlichen Möglichkeiten, die die Sprache bietet, aufzeigen.

ELAN ähnelt in den grundlegenden Konzepten den Programmiersprachen ALGOL 60, ALGOLW und ALGOL 68. Jedoch ist eine stärkere Betonung auf Sicherheit und Ausdrucksmöglichkeiten für die systematische Konstruktion von Algorithmen sowie auf das Modulkonzept gelegt worden.

1.1.1. Was nicht vorhanden ist

Nicht vorhanden sind (z.B. im Vergleich zu ALGOL60)

- GOTO-Anweisungen,
- explizite Compound Statements,
- Blöcke,
- geschachtelte Prozeduren.

Diese Sprachkonstrukte wurden fortgelassen, um die Sprache einfacher und sicherer zu gestalten.

1.1.2. Kurze Übersicht

a. Einfache Datentypen: In ELAN existieren die "denotierbaren" Datentypen:

   BOOL, REAL, INT, TEXT

   Denotierbare Datentypen sind solche, bei denen die Wertepräsentation (ELAN: Denoter; in anderen Programmiersprachen: Literal) eindeutig vorgeschrieben ist.

b. Operatoren: Für Datenobjekte dieser Typen existieren die notwendigen Operatoren (":=" ist ebenfalls Operator).

c. Deklaration: Alle Objekte müssen deklariert werden.
   Zu der bekannten Deklaration und optionalen Initialisierung aus anderen Programmiersprachen wird zusätzlich das "Accessattribut" angegeben, nämlich

   VAR für Lese-/Schreibrecht und *CONST* nur für Leserecht:

   INT CONST eins :: 1; (* *nur Lesen* *)
   REAL VAR alpha, beta :: 2.7, gamma (* *Lesen/Schreiben* *)

d. Anweisungen (Einheiten): Jede Anweisung liefert einen Wert ("expression language"), der auch "void" sein kann. "void" ist ein ausgezeichneter Wert, der "nichts" bedeutet. Anweisungen, die "void" liefern, entsprechen Statements in anweisungsorientierten Programmiersprachen ("statement languages"). Dies ermöglicht eine systematischere Behandlung von allen Sprachelementen. Setzt sich eine Anweisung aus mehreren einzelnen Anweisungen zusammen (vergleiche nächsten Punkt), so liefert sie den Wert ihrer letzten ausgeführten Anweisung.

e. Kontrollstrukturen: Außer der Sequenz werden alle Kontrollstrukturen durch Schlüsselworte eingefaßt, so daß keine expliziten Compound Statements notwendig sind.

- Sequenz

Das Symbol ";" trennt zwei Anweisungen voneinander ("statement separator") und hat die Bedeutung von "führe nächste Einheit aus".

- Abfrage

```
IF ... THEN ... END IF   oder
IF ... THEN ... ELSE ... END IF
```

- Abfrageketten

```
IF ... THEN ...
   ELIF ... THEN ... ELSE ... END IF
```

- Auswahl

```
SELECT ... OF
   CASE 1, 3: ...
   CASE 9, 5: ...
   CASE 2   : ...
   OTHERWISE ...
END SELECT
```

- Wiederholungen

```
REP ... END REP (* Endlosschleife *)
WHILE ... REP ... END REP (* abweisende Schleife *)
REP ... UNTIL ... END REP (* nichtabweisende Schleife *)
FOR a FROM  b ─UPTO─ c REP ... END (* Zaehlschleife *)
              └DOWNTO┘
```

oder Kombinationen dieser Konstruktionen.

- Refinements dienen zum "top-down"-Problemlösen im "Kleinen" (wie wir später noch erklären werden):

```
problemloesung.

problemloesung:
        teil a;
        teil b;
        teil c.
```

```
teil a:
        teil a1;
          :
          :
        teil an.
teil b:
        teil b1;
          :
          :
        teil bn.
```

usw.

Refinements besitzen keinen eigenen Datenraum (etwa wie Prozeduren) und können nicht parametrisiert werden. Deshalb und weil Refinements vom ELAN-Compiler in line ("in situ") compiliert werden, können Refinements nicht rekursiv sein.

- Terminatoren (LEAVE) beenden benannte Algorithmen: Refinements, Prozeduren, Operatoren.

```
LET n = 50;
ROW n ROW n INT VAR matrix;
   lies werte der matrix;
   lies gesuchten wert;
   IF wert gefunden THEN ... END IF;
     :
     :
wert gefunden:
INT VAR zeilenzaehler;
  FOR zeilenzaehler FROM 1 UPTO n REP
      durchsuche spalte
  END REP;
FALSE.

durchsuche spalte:
INT VAR spaltenzaehler;
  FOR spaltenzaehler FROM 1 UPTO n REP
  IF matrix [zeilenzaehler] [spaltenzaehler] = wert
      THEN LEAVE wert gefunden WITH TRUE
  END IF
  END REP
```

f. <u>Zusammengesetzte Daten</u>: Das Zusammensetzen einzelner Datenobjekte zu Datenverbunden und diese dann als Gesamtheit zu behandeln, ist in ELAN möglich. Während bei Reihungen eine feste Anzahl von Datenobjekten desselben Typs auftreten, können bei Strukturen eine feste Anzahl von Datenobjekten unterschiedlichen Typs verwendet werden. Zu beachten ist dabei, daß jede unterschiedliche Reihung bzw. Struktur ein eigener Datentyp ist, der mit keinem anderen gleich ist (Typenschutz).

Durch Subskription wird auf Einzelelemente von Reihungstypen zugegriffen:

```
ROW 10 INT VAR messwerte;
   :
   :
messwerte [3] := ...
```

Durch Selektion wird auf die Elemente eines Strukturtypes zugegriffen:

```
STRUCT (TEXT name, REAL gehalt) VAR personalbogen;
   :
   :
personalbogen.name := "hugo"
```

Denoter können für Reihungen und Strukturen gebildet werden.

"row display" für Reihungen:

```
ROW 5 INT CONST tabulator :: [1, 10, 16, 35, 71]
```

Konstruktor für Strukturen:

```
STRUCT (TEXT name, REAL gehalt) VAR karte ::
STRUCT (TEXT name, REAL gehalt): ("hahn", 1000.0);
(* besser: *)
LET KARTEIKARTE = STRUCT (TEXT name, REAL gehalt);
KARTEIKARTE VAR karte :: KARTEIKARTE: ("hahn", 1000.0)
```

g. Abkürzungsvereinbarungen erlauben die Benennung von Denotern und Datentypen innerhalb eines Moduls (PACKET).

```
LET n = 50;
ROW n INT VAR alpha; (* Anwendung *)
LET PERSON = STRUCT (TEXT name, vorname, INT alter);
PERSON VAR rainer (* Anwendung *)
```

h. Prozeduren. Es sind nur parallel deklarierte Prozeduren möglich. Parameter müssen mit Accessrecht angegeben werden:

```
PROC prozedur (INT VAR par1, (* Ein-/Ausgangs-Parameter *)
               REAL CONST par2, (* Eingangs-Parameter *)
               ROW 50 INT VAR par3):
```

Man kann Prozeduren als Parameter übergeben. Eine Prozedur darf Werte beliebigen Typs liefern:

```
INT PROC integer funktionsprozedur (...): ...

STRUCT (TEXT name, REAL gehalt)
        PROC liefert eine karteikarte: ...
```

Prozeduren (und Operatoren) sind potentiell generisch. D.h. Prozeduren dürfen gleiche Namen besitzen, müssen sich aber durch die Anzahl und/oder Typ der Parameter sowie ihre Reihenfolge unterscheiden. Die Deklaration von

```
PROC put (INT CONST wert):
:
:
END PROC put
```

und

PROC put (REAL CONST wert):
:
:
END PROC put

ist daher möglich. Der Compiler wählt die passende Prozedur nach den Angaben an der Aufrufstelle aus.
Mit

REAL VAR reelle zahl :: 3.0; INT VAR ganze zahl :: 1

und

put (1); put (ganze zahl)

wird die erste Prozedur zur Ausführung ausgewählt, während mit

*put (3.0 * reelle zahl)*

selbstverständlich die zweite ausgewählt wird.

i. Module (Packets) dienen der Zusammenfassung von Datenobjekten und Algorithmen. Eine Schnittstelle definiert, welche Bezeichner von Datenobjekten, Datentypen und Algorithmen nachfolgenden Programmteilen bekannt sein werden. Diese Objekte können dort verwendet werden, ohne daß ihre Realisierung bekannt ist (Abstraktion, insbesondere: abstrakte Datentypen).

PACKET stackhandling DEFINES pop, push:
INTEGER stackpointer :: 1;
ROW 1000 INT VAR stack;

PROC push (INT CONST wert):
:
:
END PROC push;

PROC pop (INT VAR wert):
:
:
END PROC pop

END PACKET stackhandling

Pakete können vorübersetzt und anderen Benutzern zugänglich gemacht werden. So sind z.B. die Standardfunktionen (sin, cos ... und TEXT-Operatoren u.a.m.) überwiegend in ELAN geschrieben und werden in einer vorübersetzten Form allen Benutzern zugänglich gemacht. Daher sind die Standardoperatoren und -funktionen beliebig erweiterbar.

j. Abstrakte Datentypen ("starke Datenabstraktion") und neudefinierte Operatoren.

Neudefinierte Datentypen und Zugriffsalgorithmen (neudefinierte Operatoren, Prozeduren) in Verbindung mit dem Paket-Konzept ermöglichen dem Benutzer die Schaffung neuer Sprach- bzw. Abstraktionsebenen:

```
PACKET komplex DEFINES COMPLEX, complex null, put, +:

TYPE COMPLEX = STRUCT (REAL re, im);

COMPLEX CONST complexnull :: COMPLEX: (0.0, 0.0);
(* Konstante *)

COMPLEX OP + (COMPLEX CONST links, rechts):
COMPLEX : (links. re + rechts. re,
           links. im + rechts. im)
END OP +;

PROC put (COMPLEX CONST compl):
     put (compl. re);
     put ("I");
     put (compl. im)
END PROC put

END PACKET komplex
```

1.2. Überblick über die Entwicklung der Programmiersprache ELAN *)

Die erste provisorische Beschreibung einer geplanten Sprachfamilie mit dem Namen SLAN wurde Ende 1974 von C.H.A. Koster (damals an der Technischen Universität Berlin) vorgelegt. Die Idee war die Entwicklung ververschiedener aufeinander aufbauender Sprachstufen, deren Einsatzbereich von der Ausbildung in Schulen und Universitäten bis zur Systemprogrammierung reichen sollte.

Dieser Sprachentwurf wurde durch Erfahrungen beeinflußt, die sowohl aus der Durchführung der Lehrveranstaltung "Algorithmen" im Informatik-Grundstudium, als auch aus Fortbildungskursen mit Lehrern gewonnen wurden. In diesen beiden - von sehr verschiedenen Interessenten besuchten - Veranstaltungszyklen zeigte es sich, daß die Entwicklung einer neuen Programmiersprache gerechtfertigt ist, um mit deren Hilfe mehr als bisher üblich, systematische Programmentwicklung zu unterstützen.

Einfluß auf den ersten Entwurf hatten weiterhin Diskussionen mit Mitarbeitern vom Hochschulrechenzentrum (HRZ) der Universität Bielefeld (Hahn, Heinrichs, Liedtke, Nolting), die einen eigenen Sprachentwurf vorlegten. Es wurde vereinbart, die Ausbildungsstufe von SLAN gemeinsam weiterzuentwickeln und zu implementieren. Für diese Implementierung wurden kurze Zeit später in Bielefeld zwei Diplomarbeiten an J. Liedtke und U. Bartling vergeben, aus denen der z.Zt. verfügbare ELAN-Compiler entstand.

Die Sprachfamilie SLAN wurde 1975 in einem Werkstattgespräch über Schulsprachen beim Forschungs- und Entwicklungszentrum für objektivierte Lehr- und Lernverfahren GmbH (FEoll) in Paderborn erstmals vorgestellt und in einem vom Bundesministerium für Forschung und Technologie (BMFT) geförderten Arbeitskreis Schulsprache (ASS) eingebracht. Schon vor der Konstituierung dieses Arbeitskreises konnte P. Heyderhoff vom Informatik-Kolleg der GMD (Gesellschaft für Mathematik und Datenverarbeitung) für eine Unterstützung der SLAN-Entwicklung gewonnen werden. Auch er verzichtete auf die Weiterentwicklung eines eigenen Entwurfs (SOL).

---

*) Dieser Abschnitt wurde (leicht modifiziert) aus der ELAN-Sprachbeschreibung übernommen.

SLAN wurde daraufhin im ASS von den Mitgliedern S. Jähnichen und W. Koch (TU Berlin) zusammen mit Herrn P. Heyderhoff vertreten. Die Berliner Vertreter entwickelten zusammen mit K. Kleine (ebenfalls TU Berlin) eine erste Sprachbeschreibung der inzwischen in ELAN umbenannten Schulsprache. Nicht zuletzt dadurch konnten auch andere Mitglieder des ASS von den Vorteilen dieser Sprache überzeugt werden.

Die Entwicklung einer Schulsprache wurde außerdem finanziell von der Deutschen Forschungsgemeinschaft (DFG) unterstützt. Dadurch war es möglich, an der TU Berlin Mitarbeiter hauptberuflich mit der Fortsetzung der ELAN-Aktivitäten zu betrauen. Hier müssen vor allem G. Hommel und J. Jäckel für die Weiterentwicklung der Sprache und A. Bernatzik für die Erstellung eines Compilers erwähnt werden. Wegen Auslaufens der Förderung durch die DFG Ende 1977 konnten die Compileraktivitäten allerdings nicht zu Ende geführt werden. Sie haben aber wichtige Erkenntnisse über die Implementierbarkeit einiger Sprachkonstrukte eingebracht.

Noch während der Tätigkeit des ASS konnte von dem HRZ der Universität Bielefeld eine Implementierung auf der Rechenanlage SIEMENS 4004 unter BS 1000 vorgelegt werden, die in der Implementierungssprache CDL programmiert ist und wegen der guten Portierungseigenschaften von CDL auch auf IBM 370 unter VM/370-CMS und auf der TR 440 zur Verfügung gestellt werden konnte. Die implementierte Sprache SLAN3B wich allerdings in einigen Punkten von ELAN ab. Inzwischen ist der Compiler auf ELAN umgestellt worden.

Der ASS empfahl in seinem Abschlußbericht, auf die Verwendung von BASIC zu verzichten, und empfahl stattdessen ELAN und PASCAL als Schulsprachen.

ELAN wurde dann in größerem Umfang bei der Durchführung der Informatik-Grundausbildung an der TU Berlin, in Kursen des HRZ der Universität Bielefeld und im Informatik-Kolleg der GMD Bonn eingesetzt. Weiterhin konnten in Bielefeld, Berlin und Bonn Lehrer gewonnen werden, die bereit waren, ELAN im Schul-Unterricht einzusetzen. Alle diese Aktivitäten zeigten,daß ELAN für den Anfängerunterricht in Informatik hervorragend geeignet ist.

Die Weiterentwicklung von ELAN aufgrund der praktischen Erfahrungen wurde weiterhin an der TU Berlin zusammen mit dem HRZ Bielefeld und der GMD Bonn durchgeführt. Ein Ergebnis dieser Zusammenarbeit ist die hier vorliegende Sprachbeschreibung.

ELAN ist inzwischen nicht nur auf den Rechenanlagen SIEMENS 4004, IBM 370 und TR 440 implementiert, sondern es gelang dem HRZ Bielefeld mit Unterstützung der GMD Bonn auch eine Übertragung auf einen Mikrocomputer Z80. Dabei wurde das auf ELAN zugeschnittene Betriebssystem EUMEL entwickelt. Übertragungen dieses Systems auf Mikrocomputer anderer Hersteller sind geplant.

An der TU Berlin wird zur Zeit im Rahmen des 3. DV-Programms der Bundesregierung ein Compiler für ELAN mit besserer Struktur und besseren Portierungseigenschaften fertiggestellt. Parallel dazu wird ein auf die Bedürfnisse des Einsatzes von ELAN abgestimmtes Benutzersystem entwickelt. Beide Komponenten sind zur Übertragung auf Kleinrechner vorgesehen.

1.3. Entwurfskriterien der Programmiersprache ELAN

Aus der "Beschreibung der Programmiersprache ELAN" (vergl. Kap. 1 der "Beschreibung ...") können wir folgende Entwurfskriterien herauslesen:

- ELAN soll als "Werkzeug im Informatik-Unterricht für Anfänger" verwendet werden.
- Algorithmen sollen "einfach und lesbar formuliert" werden und den "Lösungsweg widerspiegeln".
- "Die einfache Konstruktion sehr unterschiedlich strukturierter Objekte" muß möglich sein.
- Die Sprache soll "einfach in ihrer Struktur und damit in Syntax und Semantik leicht lehr- und lernbar" sein.
- "Starke Flexibilität und damit die Anpassung an alle möglichen Anwendungsgebiete" soll durch die "Anwendung weniger, ebenfalls einfacher Kompositionsregeln" erreicht werden.
- Die Sprache soll "sicher sein", so daß "ungewollte semantische Nebeneffekte möglichst vermieden werden".
- Die Programmiersprache soll die "systematische Konstruktion von Algorithmen mit der top-down Methode" (Programmieren im Kleinen) sowie die bottom-up Methode (Programmieren im Großen) unterstützen.
- Die Programmiersprache sollte insbesondere "die Gelegenheit bieten, alle bei der Entwicklung des Programms entstandenen Abstraktionen explizit im Programmtext zu belassen".

Das Hauptanwendungsgebiet der Programmiersprache ELAN liegt also auf der Ausbildung. ELAN ist daher keine "Allzwecksprache". Wichtige bekannte Sprachkonzepte wurden absichtlich nicht mit in die Programmiersprache aufgenommen.

Ob diese Entwurfskriterien in der Sprache verwirklicht wurden, bleibt der Kritik des Lesers und Benutzers der Programmiersprache ELAN überlassen. Nach unserer Meinung muß jedoch bedacht werden, daß auch neue Programmiersprachen nicht in jeder Hinsicht "vollkommen" sein können und bei der Entwicklung immer Kompromisse geschlossen werden müssen. Eine kritische Haltung gegenüber Programmiersprachen ist daher nicht nur erwünscht, sondern notwendig.

2. Die lexikalischen Elemente

In diesem Kapitel werden wir die lexikalischen Elemente von ELAN, d.h. die Elemente, in denen ein ELAN-Programm notiert wird, beschreiben. Es existieren folgende lexikalische Elemente in ELAN:

- Zeichenfolgen für bestimmte Worte, die eine feste Bedeutung haben (sogenannte "reserved symbols"),
- Zeichenfolgen zur Bezeichnung von neudefinierten Datentypen und Operatoren (sogenannte "bold tags"),
- Bezeichner,
- Sonderzeichen,
- Denoter und
- Kommentare.

Die Repräsentation, d.h. die Darstellung dieser lexikalischen Elemente in Programmen, ist prinzipiell nicht festgelegt, so daß die Sprache ELAN auch mit stark voneinander abweichenden Zeichensätze implementiert werden kann. In diesem Buch werden die in der "Beschreibung der Programmiersprache ELAN" empfohlenen Repräsentationen verwendet.

2.1. Schlüsselwörter

In der ELAN-Sprachbeschreibung wird eine Unterscheidung zwischen der Repräsentation von Namen (Bezeichnern), den Namen von neudefinierten Datentypen und Operatoren ("bolds") und Wörtern mit einer festen Bedeutung ("reserved") getroffen. Der Grund für diese (syntaktische) Unterscheidung ist, daß Wörter mit einer festen Bedeutung, wie z.B. WHILE, UNTIL usw. nicht als Namen für neudefinierte Operatoren und Datentypen verwendet werden dürfen.

Prinzipiell könnten diese drei Arten durch drei verschiedene Zeichensätze dargestellt werden, die jedoch kaum auf einer Rechenanlage gleichzeitig zur Verfügung stehen dürften. Darum werden nur

- Namen und
- Worte mit einer festen Bedeutung sowie Namen für neudefinierte Datentypen und Operatoren

in Programmen unterschiedlich dargestellt.

Die letzteren bezeichnen wir in diesem Buch in Anlehnung an andere Programmiersprachen als Schlüsselwörter.

Steht ein Zeichensatz mit großen und kleinen Buchstaben zur Verfügung, dann können Schlüsselworte groß geschrieben werden. In diesem Fall müssen Bezeichner mit kleinen Buchstaben (und Ziffern) notiert werden.

Beispiel:

INT VAR bezeichner

Steht ein ELAN-Übersetzer zur Verfügung, der einen solchen Zeichensatz verarbeiten kann, so ist es möglich, Bezeichner als Schlüsselworte (und umgekehrt) zu verwenden, sofern nur die richtigen Zeichen gewählt werden, weil die Unterscheidung durch die Wahl von großen und kleinen Buchstaben erfolgt.

Steht jedoch ein Rechnersystem zur Verfügung, daß nur einen Zeichensatz mit großen Buchstaben verarbeiten kann, so können Bezeichner nicht wie Schlüsselwörter geschrieben werden. Das bedeutet, daß Schlüsselwörter reserviert sind und nicht anders verwendet werden dürfen. Eine andere Möglichkeit ist es, Schlüsselwörter durch Hochkommata zu kennzeichnen. Diese Möglichkeit ist jedoch anfällig gegen Schreibfehler (vergessene Hochkommata) und sollte deshalb vermieden werden.

Beispiel:

'INT' 'VAR' BEZEICHNER

2.2. Bezeichner (Namen)

Bezeichner ("identifier") werden verwendet, um Objekte in einem Programmtext zu benennen und zu identifizieren. Die Länge von Bezeichnern unterliegt keiner (praktischen) Einschränkung. Weiterhin dürfen - zur besseren Lesbarkeit - Leerzeichen in Bezeichnern verwendet werden, die aber nicht Teil des Bezeichners sind.

Beispiel:

das ist ein name

dasisteinname

Beide Zeichenfolgen stellen den gleichen Bezeichner dar, d.h. Leerzeichen ("blanks") werden von einem ELAN-Übersetzer überlesen. Bezeichner werden immer von Schlüsselworten, Sonderzeichen oder Kommentaren eingeschlossen, so daß sie immer eindeutig sind.

Bezeichner bestehen aus mindestens einem Buchstaben. Dieser Buchstabe kann von weiteren Buchstaben und/oder Ziffern gefolgt werden. Es entspricht guter Programmierpraxis, immer sinnvolle Bezeichner in Programmen zu verwenden. Oft ist jedoch das Finden von Bezeichnern beinahe ebenso schwierig wie die Konstruktion des Algorithmus selbst und sollte deshalb nicht unterschätzt werden.
Der Unterschied zwischen einem Namen und einem Bezeichner liegt darin, daß ein Bezeichner immer nur ein Objekt kennzeichnet, während ein Name für mehrere unterschiedliche Objekte unter bestimmten Umständen verwendet werden kann (vergl. Kap. 11.2.2). In diesem Buch sprechen wir oft von dem Namen eines Datenobjekts oder einer Prozedur, wenn wir nicht auf den Gültigkeitsbereich dieses Namens oder auf den Identifizierungsvorgang eingehen wollen.

2.3. Sonderzeichen

Sonderzeichen, d.h. Zeichen, die keine (großen oder kleinen) Buchstaben oder Ziffern sind, werden in ELAN zur Repräsentation von Operatoren, zur Trennung von Anweisungen usw. verwendet. Wir werden die Bedeutung von Sonderzeichen im Zusammenhang mit der Erklärung von Sprachelementen besprechen.

2.4. Kommentare

Kommentare können (fast) an jeder beliebigen Stelle eines ELAN-Programms eingefügt werden. Verboten ist jedoch die Verwendung von Kommentaren innerhalb lexikalischer Einheiten bei Schlüsselworten, Namen usw. Kommentare haben keinen Einfluß auf die Ausführung eines Programms und werden deshalb von einem ELAN-Übersetzer überlesen. Kommentare dienen ausschließlich der Dokumentation von Programmen, sind aber oft gar nicht notwendig, wenn geeignete Bezeichner im Programmtext verwendet werden.
Kommentare werden in Kommentar-Klammern eingefaßt, für die unterschiedliche Repräsentationen existieren:

öffnende Kommentar-Klammern	schließende
(*	*)
{	}
#(	)#

In diesem Buch werden ausschließlich die Klammern "(*" und "*)" verwendet. Die anderen Repräsentationen sind entweder nicht in jedem Zeichensatz vorhanden oder haben bei bestimmten Rechnersystemen eine besondere Bedeutung ("#" ist Fluchtsymbol auf der TR 440).

Innerhalb von Kommentaren darf jedes verfügbare Zeichen verwendet werden. Es ist möglich, innerhalb von Kommentaren wieder einen Kommentar zu schreiben (verschachtelte Kommentare). Mit geschachtelten Kommentaren kann man leicht - etwa für Testzwecke - Programmteile ausblenden oder in ein Programm einfügen.

3. Die Verwendung einfacher Datentypen

In diesem Kapitel werden die einfachen ELAN-Datentypen vorgestellt. Sie sind - im Gegensatz zu anderen, noch später zu behandelnden - von der Sprache ELAN vorgegeben.

Weiterhin existieren bestimmte Formen von Werte-Repräsentationen für die einfachen Datentypen, die in ELAN Denoter genannt werden. Ein weiterer wichtiger Begriff in ELAN ist das Accessrecht. Es gibt an, ob mittels eines Bezeichners auf ein Datenobjekt "nur lesend" oder "lesend und schreibend" zugegriffen werden darf. Datenobjekte müssen in ELAN vereinbart werden (Deklaration).

3.1. Datentypen und Datenobjekte in ELAN

Ziel von Programmen ist es, Datenobjekte derart zu manipulieren, daß für eine bestimmte Eingabe ("Input") eine bestimmte Ausgabe ("Output") erzielt wird. Deshalb ist es wichtig, daß wir uns mit dem Konzept des Datenobjektes in ELAN befassen.

Datenobjekte in ELAN haben bestimmte Eigenschaften, deren Kenntnis auch wichtig für das Verstehen von Programmen ist:

a) Wert
Datenobjekte besitzen Werte. Diese Werte müssen von einer bestimmten Art (Typ) sein.

b) Datentyp
Der Datentyp bestimmt die Art der Werte, die ein Datenobjekt annehmen kann. In ELAN gibt es verschiedene Klassen von Datentypen:

- die einfachen Datentypen, die auch primitive Datentypen oder klassisch denotierbare Typen genannt werden,
- die Reihungstypen, die mehrere gleichartige Datentypen zusammenfassen,
- die Strukturtypen, die ungleichartige Datentypen zusammenfassen,
- die neudefinierten Datentypen, mit denen Programmierer neue Typen aus bereits bestehenden konstruieren und somit auf einer abstrakteren Ebene arbeiten können.

c) Speicherplatz
Ein Datenobjekt hat einen bestimmten Speicherplatz etwa im Zentralspeicher des Rechners, auf dem das Programm ausgeführt ("elaboriert") werden soll. Für unterschiedliche Datentypen werden im allgemeinen unterschiedlich große Speicherplätze benötigt.

Datenobjekte können gelesen und/oder durch Operationen verändert werden. Eine Vereinbarung (Deklaration) verbindet ein Datenobjekt mit einem Namen und legt den Datentyp und das Accessrecht fest. Eine Initialisierung kann dem Datenobjekt bei der Deklaration einen Wert geben.

Erfolgt keine Initialisierung, so hat das Datenobjekt einen undefinierten Wert.

3.2. Die einfachen Datentypen

Die einfachen (primitiven, klassisch denotierbaren, elementaren) Datentypen sind:

a) INT
Bezeichnet eine Untermenge der ganzen Zahlen.

b) REAL
Bezeichnet eine Untermenge der reellen Zahlen.

c) BOOL
Bezeichnet Wahrheitswerte.

d) TEXT
Bezeichnet Zeichen und Zeichenfolgen.

3.3. Denotierungen der einfachen Datentypen

Für die einfachen Datentypen existieren vorgeschriebene Formen der Darstellung von Werten in Programmen, die in ELAN Denoter genannt werden. Durch die Darstellung eines Denoters wird implizit auch sein Datentyp festgelegt. Das Accessrecht von Denotern ist immer CONST (diesen Begriff erklären wir im nächsten Abschnitt).

Der darstellbare Wertebereich von INT- und REAL-Datentypen und damit von deren Denotern, sowie die Anzahl der Zeichen in Texten sind implementationsabhängig (vergl. Kap. 13.).

a) INT-Denoter
sind Aneinanderreihungen von Ziffern, etwa

1 17 0 007

(führende Nullen werden überlesen).

Jedoch ist beispielsweise -1 kein INT-Denoter, sondern eine Aufeinanderfolge eines monadischen Operators und eines INT-Denoters.

b) REAL-Denoter
sind zwei INT-Denoter, die durch einen Punkt getrennt und die von einem optionalen Exponenten-Teil gefolgt werden. Der Exponenten-Teil besteht aus dem Exponenten-Symbol (E) und einem optionalen Vorzeichen (kein Operator), dem ein INT-Denoter folgt.

- REAL-Denoter ohne Exponenten-Teil, etwa

  3.14 7.9999

  aber nicht

  -3.14

  weil wiederum ein monadischer Operator verwendet wurde, und auch nicht

  .007

  weil mindestens eine Ziffer vor dem Dezimalpunkt fehlt.

- REAL-Denoter mit Exponenten-Teil sind

  0.314E+1 0.314E1 31.4E-1

c) BOOL-Denoter
werden durch die Schlüsselworte

TRUE

und

FALSE

dargestellt.

d) TEXT-Denoter
sind Zeichen oder Zeichenfolgen, die durch Anführungszeichen eingeschlossen werden:

"das ist ein text"
"JETZT folgt ein leerer TEXT"
""
"und jetzt einer, der das quote symbol einmal enthaelt"
"mit "" symbol"
"und nur ein quote symbol:"
""""

Welche Zeichen in TEXT-Denotern zugelassen sind, ist implementationsabhängig. Das gilt insbesondere für die Zeichen "CR" (=carriage return) und "LF" (=line feed), sofern ein TEXT-Denoter sich über mehr als eine Zeile erstreckt.

3.4. Das Accessrecht

Wie wir schon im vorhergehenden Abschnitt erwähnt haben, werden Datenobjekte nach dem Datentyp klassifiziert. Hinzu kommt noch das Accessrecht (auch Zugriffsrecht genannt). Das Accessrecht gibt an, ob auf ein Datenobjekt mittels eines Namens

- nur lesend (CONST)
- oder lesend/schreibend (VAR)

zugegriffen werden darf. Ist also der Bezeichner eines Datenobjekts mit CONST vereinbart, dann darf er nicht an Stellen eines Konstrukts erscheinen, die einen "VAR"-Bezeichner verlangen, z.B. auf der linken Seite einer Zuweisung. Für den Bezeichner, der mit dem Accessrecht VAR vereinbart wurde, ist die Zugriffs-Regelung leider nicht so einfach. Hier hängt es davon ab, in welchem Kontext der Bezeichner für das Datenobjekt verwendet wird. Wird der Bezeichner in einem Sprachelement benötigt, das ein VAR-Accessrecht erwartet, so darf das Datenobjekt verändert werden. Erwartet dagegen ein Konstrukt einen Bezeichner mit dem Accessrecht CONST, so wird eine Anpassungsoperation vorgenommen ("consting"), so daß auf das Datenobjekt mit dem vereinbarten Accessrecht VAR nur lesend zugegriffen werden darf (vergl. Kapitel 4.6.).

Der Datentyp und das Accessrecht wird zusammen mit dem Bezeichner des Datenobjekts bei einer Deklaration (siehe nächsten Abschnitt) angegeben.

Beispiel:

INT CONST anzahl monate :: 12

vereinbart ein Datenobjekt vom Typ INT, auf das nur lesend zugegriffen, das also nicht verändert werden darf.

BOOL VAR liste durchsucht

vereinbart ein Datenobjekt vom Typ BOOL, auf das lesend und schreibend zugegriffen werden darf.

Der Begriff Accessrecht wird uns später noch bei anderen Sprachkonstrukten begegnen. Wir werden dann den Begriff, der bisher nur aus VAR und CONST besteht, erweitern müssen.

Anmerkung:
Das Accessrecht dient ausschließlich zur Überprüfung von externen Objekten, d.h. Objekten, die dem Programmierer in einem Programm zur Verfügung stehen (Namen, Sprachelemente, die Datenobjekte liefern usw.). Es hat keine direkten Auswirkungen auf interne Objekte, d.h. Objekte, die zur Laufzeit des Programms existieren. Das bedeutet, daß das Accessrecht für Überprüfungen ausschließlich zur Übersetzungszeit ("compile time") eines ELAN-Programmes herangezogen wird.

Aus Gründen der besseren Lesbarkeit und Verständlichkeit werden wir die Begriffe "VAR-Datenobjekt" bzw. "CONST-Datenobjekt" synonym für "mit VAR (bzw. CONST) deklarierte Namen eines Datenobjekts" verwenden, obschon sich das Accessrecht ausschließlich auf den Namen und nicht auf das Datenobjekt bezieht.

3.5. Die Vereinbarung von Datenobjekten (Deklaration)

Auf Datenobjekte können bestimmte Operationen (vergl. Kap. 4) angewendet werden. Die Art und der Umfang einer Operation wird durch den Datentyp und das Accessrecht bestimmt. Dadurch bietet die Sprache ELAN die Möglichkeit, fehlerhaftes Verwenden von Datentypen und dazugehöriger Operationen durch einen Übersetzer "automatisch" zu erkennen. Zu diesem Zweck wird verlangt, daß jedes Datenobjekt mit seinem Datentyp und einem Accessrecht vereinbart (deklariert) wird.

Die Deklaration eines Datenobjekts besteht aus der Angabe

- des Datentyps,
- des Accessrechts und
- des Namens (Bezeichner).

Beispiel:

"INT VAR alpha teilchen" vereinbart ein Datenobjekt mit den Namen "alpha teilchen", dessen Datentyp INT ist und das Accessrecht VAR erhält. Eine Veränderung des Wertes des Datenobjekts ist erlaubt.

"REAL CONST geholt" deklariert ein Datenobjekt vom Datentyp REAL und dem Accessrecht CONST. Somit ist keine Verwendung in Konstrukten erlaubt, die das Accessrecht VAR verlangen. Das Datenobjekt kann nur gelesen werden.

Man beachte, daß beide Datenobjekte mit dieser Form der Deklaration noch keine Werte besitzen. Das ist besonders wichtig bei dem letzten Beispiel, weil ein CONST-Datenobjekt durch eine Zuweisung keinen Wert erhalten kann. In Kapitel 4 werden wir sehen, wie Datenobjekte initialisiert, d.h. mit einem Wert bei der Deklaration versehen werden.

Durch eine Aufzählung von mehreren Namen mit gleichem Accessrecht können Datenobjekte gleichen Typs in einer Deklaration vereinbart werden. Hierbei werden der Datentyp und das Accessrecht nur einmal am Anfang aufgeführt.

INT VAR zahl, x wert, mittelwert

Deklarationen (zu denen nicht nur Deklarationen von Datenobjekten gehören, sondern auch die Vereinbarung von Prozeduren, Operatoren u.a.m.) gehören neben den Einheiten (vergl. Kap. 6) zu den Grundbausteinen der Sprache ELAN.

4. Operatoren für einfache Datentypen

In diesem Kapitel befassen wir uns mit den Operatoren für die im vorigen Kapitel eingeführten einfachen Datentypen. Dabei besprechen wir die monadischen und dyadischen Operatoren, sowie das Prinzip der Generizität. Zum Schluß gehen wir auf die Initialisierung bei der Deklaration ein.

4.1. Objekte und Elaboration

Wir haben bisher schon eine Art von Objekt in der Programmiersprache ELAN kennengelernt: das Datenobjekt. Eine andere Art von Objekten sind Operationen, die auf Datenobjekte angewendet werden.

Während Datenobjekte durch die Anwendung bestimmter Operationen verändert und/oder geliefert werden, können Operationen nur angewendet bzw. ausgeführt (elaboriert) werden.

Die Elaboration von Operationen kann also eine Änderung von Datenobjekten bewirken oder Datenobjekte liefern. Wir unterscheiden in ELAN zwischen zwei Arten von Operationen, nämlich Operatoren und Prozeduren. In diesem Kapitel besprechen wir nur die Operatoren für einfache Datentypen.

4.2. Monadische und dyadische Operatoren

In ELAN existieren eine Reihe von Operatoren, die für die einfachen Datentypen vorgesehen sind.

Es wird zwischen monadischen und dyadischen Operatoren unterschieden. Monadische Operatoren wie etwa

- 3

arbeiten nur auf einem Operanden, der rechts vom Operator-Symbol geschrieben werden muß.

Dyadische Operatoren dagegen arbeiten auf zwei Operanden. Hier muß das Operator-Symbol zwischen den Operanden stehen:

7 - 3

Wir sehen an diesen beiden Beispielen auch, daß verschiedene Operatoren (hier monadisches und dyadisches Minus) die gleiche Repräsentation besitzen, obwohl es sich um unterschiedliche Operatoren handelt. Diese Darstellung wurde sinnvollerweise aus der mathematischen Schreibweise übernommen und wird auch in anderen Programmiersprachen benutzt. Zusätzlich werden die Operatoren ebenfalls durch den Datentyp der Operanden unterschieden. So ist etwa der dyadische "+" - Operator für INT - wie auch für TEXT-Datentypen definiert, obwohl sehr unterschiedliche Operationen durchgeführt werden müssen (Addition von ganzen Zahlen bzw. Aneinanderkettung von Texten).

Dieses Prizip, daß gleiche Namen für unterschiedliche Operationen verwendet werden, wird Generizität genannt. Weil Generizität auch noch bei einem anderen Sprachelement verwendet wird (Prozeduren), spricht man von generischen Operatoren. Operatoren in ELAN werden also durch

- den Namen,
- die Anzahl der Operanden (monadisch bzw. dyadisch)
- und den Datentyp der Operanden unterschieden.

Letztendlich können die Operatoren einen Wert liefern. Zum Beispiel liefert die Operation

7 - 3

(hoffentlich) den Wert 4. Dieser ist vom Datentyp INT. Man sagt auch, daß der dyadische "-" Operator, der auf Operanden vom Datentyp INT arbeitet, ein Resultat vom Typ INT liefert. Es existieren jedoch auch Operatoren, deren Operanden unterschiedliche Datentypen besitzen und/oder deren Resultats-Datentyp unterschiedlich von dem der Operanden ist.

*2 * "ha"*

liefert "haha". Der erste (linke) Operand ist hier vom Datentyp INT, der zweite (rechte) Operand ist vom Typ TEXT, während das gelieferte Resultat wiederum vom Typ TEXT ist.

Dabei ist zu beachten, daß das Resultat eines Operators nicht zur Kennzeichnung der generischen Operatoren herangezogen wird. Das bedeutet, daß zwei Operatoren, deren Operanden gleiche Datentypen besitzen, aber unterschiedliche Resultate liefern, verboten sind.

Im folgenden werden wir einige der gängigen monadischen und dyadischen Operatoren und deren Resultat-Typ aufführen. Eine Übersicht über weitere Operatoren findet sich im Kapitel 13 (Standard-Pakete).

Wichtig ist noch folgendes: Während der Datentyp des Resultats durch den ausgeführten Operator, d.h. aufgrund der Generizität von Operatoren durch den Datentyp der Operanden, bestimmt wird, ist das Accessrecht eines Resultats immer CONST.

Von dieser Regel existiert (leider) keine Ausnahme. Aber diese Regel ist auch einleuchtend: Durch die Ausführung (Elaboration) eines Operators entsteht häufig ein neues Datenobjekt, das Resultat. Dieses Datenobjekt kann nicht über einen Bezeichner angesprochen werden, da das Resultat keinen Bezeichner besitzt. Das bedeutet, daß keine Operationen, insbesondere verändernde Operationen, die das VAR-Accessrecht voraussetzen, auf ein Resultat vorgenommen werden können.

Die Ausführung eines Operators sollte - außer dem Resultat - keine weiteren Auswirkungen haben. Insbesondere sollten der oder die Operanden nicht verändert werden.

Eine Ausnahme bildet z.B. der Zuweisungsoperator. Dieser liefert kein Resultat. Weiterhin muß der linke Operand, d.h. der Bezeichner des Datenobjekts, das auf der linken Seite einer Zuweisung erscheint, mit dem Accessrecht VAR vereinbart worden sein. Daraus geht hervor, daß kein Ausdruck (vergl. Abschnitt 4.3) auf der linken Seite einer Zuweisung geschrieben werden darf.

4.2.1. Einige monadische Operatoren

Repräsentation	Typ rechter Operand	Typ Resultat	Wirkung
NOT	*BOOL*	*BOOL*	Negation
LENGTH	*TEXT*	*INT*	Länge eines TEXTs
+	*INT* *REAL*	*INT* *REAL*	keine Wirkung " "
-	*INT* *REAL*	*INT* *REAL*	Vorzeichenwechsel "
SIGN	*INT* *REAL*	*INT* *INT*	Vorzeichen "
ABS	*INT* *REAL*	*INT* *REAL*	Betrag "

4.2.2. Einige dyadische Operatoren

In der Spalte "Wirkung" wird der linke Operand mit "l", der rechte mit "r" bezeichnet.

Repräsentation	Typ linker Operand	Typ rechter Operand	Typ Ergebnis	Wirkung
:=	*INT*	*INT*	–	Zuweisung
	REAL	*REAL*	–	"
	BOOL	*BOOL*	–	"
	TEXT	*TEXT*	–	"
=,<>,< <=,>,>=	*INT*	*INT*	*BOOL*	gleich,
	REAL	*REAL*	*BOOL*	ungleich,
	TEXT	*TEXT*	*BOOL*	kleiner, kleiner gleich, größer, größer gleich
+	*INT*	*INT*	*INT*	Addition
	REAL	*REAL*	*REAL*	"
	TEXT	*TEXT*	*TEXT*	Aneinanderfügen
–	*INT*	*INT*	*INT*	Subtraktion
	REAL	*REAL*	*REAL*	"
*	*INT*	*INT*	*INT*	Multiplikation
	REAL	*REAL*	*REAL*	"
	INT	*TEXT*	*TEXT*	Vervielfältigung
DIV	*INT*	*INT*	*INT*	Division
/	*REAL*	*REAL*	*REAL*	Division
MOD	*INT*	*INT*	*INT*	Rest aus l DIV r
	REAL	*REAL*	*REAL*	vergl. 13.3.
**	*INT*	*INT*	*INT*	Exponentation
	REAL	*INT*	*REAL*	"
INCR	*INT*	*INT*	–	l := l + r
	REAL	*REAL*	–	l := l + r
DECR	*INT*	*INT*	–	l := l - r
	REAL	*REAL*	–	l := l - r
CAT	*TEXT*	*TEXT*	–	l := l + r
SUB	*TEXT*	*INT*	*TEXT*	Teiltext (1 Zeichen)
AND, OR, XOR	*BOOL*	*BOOL*	*BOOL*	(logisches)und bzw. oder, exkl. oder

4.3. Ausdrücke (Formeln) und Prioritäten

Häufig treten nicht nur einzelne Operatoren und Operanden auf, sondern mehrere, etwa

*3+4*4*

Eine solche Aneinanderreihung von Operationen nennt die Beschreibung eine Formel ("formula"), deren einfachster Fall ein dyadischer oder monadischer Operator mit entsprechenden Operanden bzw. der Name für ein Datenobjekt oder ein Denoter ist.
Hierbei ist es wichtig, daß die Reihenfolge der Elaboration der Operatoren festgelegt wird. Würde man nämlich zuerst 3+4 errechnen und dieses Ergebnis mit 4 multiplizieren, dann erhält man ein anderes Ergebnis, als wenn erst 4*4 berechnet und darauf der Wert von 3 addiert wird.

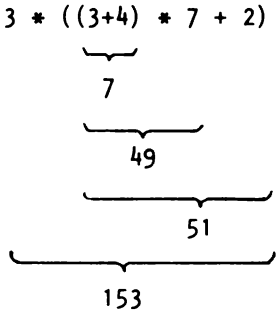
Daher ordnet die ELAN-Beschreibung (im Gegensatz zu einigen anderen Programmiersprachen) den Operatoren Prioritäten zu, die die Reihenfolge der Elaboration festlegen. Dabei wird der Operator mit höherer Priorität vor einem Operator mit niedrigerer Priorität elaboriert.

Priorität	Operator(en)
9	alle monadischen Operatoren (insbesondere: NOT,+,-)
8	**
7	*,/, DIV,MOD
6	(dyadisches)+,-
5	=,<>,<,<=,>,>=
4	AND
3	OR
2	alle dyadischen Operatoren, außer denen, die in dieser Tabelle angeführt sind.
1	:=

Das bedeutet, daß etwa die Zuweisung (":=") immer zuletzt ausgeführt wird, was man ja auch erwarten sollte.

Bei Formeln, in denen Operatoren gleicher Priorität auftreten, werden diese von links nach rechts ausgeführt. Jedoch werden die Operanden eines Operators in nicht definierter Reihenfolge ausgewertet. Das ist besonders dann wichtig, wenn einer der Operanden eines Operators aus einem Prozeduraufruf besteht, der z.B. über einen Seiteneffekt den anderen Operanden verändert. Schließlich können Ausdrücke bzw. Formeln geklammert werden. Dabei werden die Formeln, die eingeklammert sind, zuerst ausgewertet. Das gilt auch für geschachtelte Klammerausdrücke.

Beispiel:



Reihenfolge der Auswertung unter Berücksichtigung der Prioritäten und Klammerung

Wichtig ist ferner der Datentyp des Datenobjekts, das eine Formel liefert. Dies ist das Resultat des letzten Operators, der unter Beachtung der Prioritäten, der Auswertung von links nach rechts und der Klammern elaboriert wird.

Beispiel:

*3 * 2 * "ha"*

liefert einen TEXT (CONST)-Datentyp, weil zuerst 3*2 (Operator für INT's, liefert ein INT-Typ) und dann der "Vervielfältigungs"-Operator für Texte elaboriert wird.

Wie wir schon bei Operatoren erwähnt habne, hat das Resultat einer Formel (d.h. die Anwendung eines oder mehrerer Operatoren) das Accessrecht CONST.

4.4. Die Zuweisung

Die Zuweisung ("assignation") ist eine besonders wichtige Operation, weil sie kein Resultat liefert, jedoch als Wirkung den Wert des einen Operanden verändert. Der Operator benötigt zwei Operanden, dessen linker mit dem Accessrecht VAR, der rechte mit CONST versehen sein muß. Das Accessrecht VAR (Lese/Schreibrecht) für den linken Operanden wird benötigt, damit an diesen Operanden der Wert des rechten zugewiesen werden kann. Meist findet ein Wertetransport statt. Es ist wichtig, daß die beiden Operanden den gleichen Datentyp besitzen. Dabei darf der rechte Operand aus einer Formel bestehen, die aber ein Objekt vom gleichen Datentyp wie der linke Operand liefern muß.

Beispiel:

```
INT VAR ergebnis, zahl;
zahl := 3;
ergebnis := 2*zahl
```

Wird ein Zuweisungsoperator neu definiert (vergl. Kap. 10.2.), z.B. für neue Datentypen, so kann er auch anders als durch einen Wertetransport realisiert werden. Man kann auch Zuweisungsoperatoren für Operanden ungleichen Datentyps definieren.

4.5. Deklaration mit Initialisierung

Wie wir in Abschnitt 3.5. erklärt haben, muß jedes Datenobjekt mit seinem Namen, Typ und Accessrecht deklariert werden. Die so deklarierten Datenobjekte haben aber noch einen undefinierten Wert. Einen Wert können mit dem Accessrecht VAR vereinbarte Namen von Datenobjekten, nach dem bisher gesagten, erst mit einer Zuweisung erhalten.

ELAN bietet auch die Möglichkeit, gleichzeitig mit der Deklaration einem Datenobjekt einen Wert zu geben. Diesen Vorgang nennt man Initialisierung.

Beispiel:

```
INT VAR index :: 1;
REAL CONST pi :: 3.14159
```

Die Initialisierung entspricht einer Zuweisung nach der Deklaration, hingegen wird anstatt des Zuweisungs-Symbols (":=") das "initial symbol" ("::") verwendet, um die Zuweisung von der Initialisierung zu unterscheiden. Wichtig bei dem Vorgang der Initialisierung ist, daß sie jedesmal durchgeführt wird, wenn sie im Programm durchlaufen wird (auch bei CONST-Datenobjekten!), während die eigentliche Deklaration nur bei Eintritt in den jeweiligen Gültigkeitsbereich ("range") elaboriert wird. Darauf werden wir nochmals etwas später mit Beispielen eingehen (vergl. Kap. 6).

Die Initialisierung eines mit dem Accessrecht CONST deklarierten Namens eines Datenobjekts ist die einzige Möglichkeit, diesem einen Wert zu geben. Bekanntlich können wir einen mit CONST vereinbarten Namen eines Datenobjekts nicht als linken Operanden eines Zuweisungs-Operators verwenden, d.h. wir können an ihn nicht zuweisen.

Mit anderen Worten: mit CONST vereinbarte Namen von Datenobjekten müssen initialisiert werden, während VAR-Datenobjekte erst später Werte (etwa durch eine Zuweisung) erhalten können.

Welche Art von Ausdrücken darf nun auf der rechten Seite des Initialisierungs-Symbols stehen? Das sind alle diejenigen Konstrukte, die den deklarierten Typ liefern, d.h. auch Formeln (Ausdrücke).

Beispiel:

```
INT VAR alpha :: 3,
        beta  :: 4*alpha,
        gamma :: (beta+3)*5;

INT CONST delta :: sign (x) (* wobei x zuvor eingelesen wurde *)
```

Weitere Konstrukte, die auf der rechten Seite des Initialisierungs-Symbols erscheinen dürfen, werden in folgenden Kapiteln besprochen. Es ist jedes Konstrukt erlaubt, das denselben Datentyp liefert. Dabei müssen die Werte von Datenobjekten, die auf der rechten Seite verwendet werden, definiert sein.

4.6. Wandlung des Accessrechts

Im Gegensatz zu vielen anderen Programmiersprachen existieren in ELAN keine impliziten Typenanpassungen (z.B. INT->REAL). Das hat u.a. zur Konsequenz, daß in einem Ausdruck, der einen REAL-Wert liefern soll, kein INT-Denoter verwendet werden darf.

Beispiel:

REAL CONST drei :: 3.0 ('3' waere falsch *)*

Zwei wichtige Gründe unterstützen das Verbot von impliziten Anpassungen:

- Der Schutz durch die Überprüfung von Datentypen in Programmen wird erhöht.
- Es können keine für Programmierer unsichtbaren Ineffizienzen durch Anpassungsoperationen in Programmen auftreten.

Nachteilig ist, daß die Flexibilität eingeschränkt wird. Es existieren jedoch explizite Operationen (Prozeduren), mit denen Datentyp-Wandlungen vorgenommen werden können. Zudem können Programmierer jederzeit neue Operationen schreiben (Prozeduren und Operatoren), die bestimmte implizite Typanpassungen vornehmen können.

Die einzige Operation, die implizit vorgenommen wird, betrifft das Accessrecht. Dabei existieren zwei Operationen, das "consting" und die Balancierung des Accessrechts.

4.6.1. "consting"

Wie schon in Abschnitt 4.3 erwähnt, kann ein Operator als Resultat ein Datenobjekt mit dem Accessrecht CONST liefern. Eine solche Formel kann auf der rechten Seite einer Zuweisung erscheinen:

INT VAR zahl1 :: 1, zahl2 :: 2, summe;
summe := zahl1 + zahl2

Das bedeutet, daß für den linken Operand des Zuweisungsoperators das Accessrecht VAR, für den rechten Operand das Accessrecht CONST erwartet wird.

Was passiert nun, wenn als rechter Operand ein Objekt mit dem Accessrecht VAR geschrieben wird?

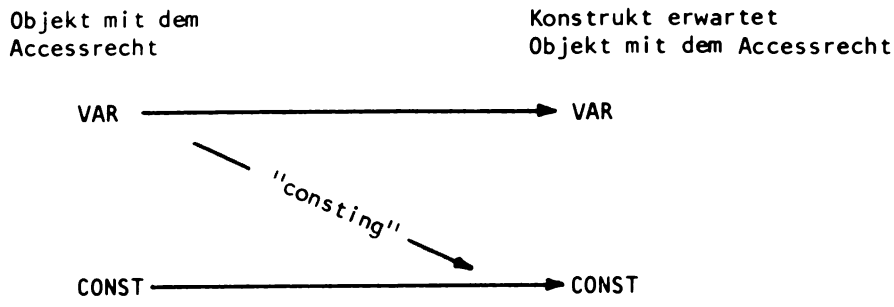
Beispiel:

INT VAR zahl1 :: 1, ergebnis;
ergebnis := zahl1

Offensichtlich muß das Accessrecht des Namens "zahl1" von VAR nach CONST gewandelt werden, da der Zuweisungsoperator als rechten Operanden ein CONST-Objekt erwartet. Nur die Wandlung des Accessrechts von VAR (lesen/schreiben) nach CONST (nur lesen) ist sinnvoll, da das Datenobjekt durch diese Reduzierung der erlaubten Operationen nicht gefährdet wird, sozusagen einen zusätzlichen Schutz erfährt. Die Wandlung des Accessrechts wird "consting" genannt.

Eine (theoretisch mögliche) Wandlung des Accessrechts von CONST nach VAR ist nicht sinnvoll und daher in ELAN verboten, da hierdurch ein explizit geforderter Schreibschutz zunichte gemacht würde.

Diesen Sachverhalt kann man graphisch folgendermaßen darstellen:



Anmerkung: Wie schon in Abschnitt 3.4. erwähnt, sprechen wir in diesem Buch oft von VAR- bzw. CONST-Datenobjekten. Dies ist zwar nicht korrekt, denn das Accessrecht ist an den Bezeichner und nicht an ein Datenobjekt gebunden, doch dieser Unterschied spielt für Programmierer kaum eine Rolle, so daß wir bei diesen (bequemen) Begriffen bleiben wollen.

In ELAN existiert noch ein weiteres Accessattribut, das "proc PARAMETY"-Attribut. Es dient vorwiegend der syntaktischen Beschreibung von formalen Prozeduren und hat für Programmierer keine große Bedeutung. Ein solches Accessattribut kann nach CONST gewandelt werden. Diese Wandlung wird "deproceduring" genannt. Wir gehen auf diesen Sachverhalt etwas später ein.

4.6.2. Balancierungen des Accessrechts

Es existieren einige Sprachelemente in ELAN, bei denen zwei oder mehr Teile potentiell einen Wert liefern können. Liefern diese Teile Objekte mit unterschiedlichem Accessrecht, so muß das Accessrecht "balanciert" werden, so daß das gesamte Konstrukt ein Objekt mit einem einheitlichen Accessrecht liefert. Dies ist dann immer CONST. Wir werden die Balancierung bei der Besprechung dieser Sprachelemente (Abfrage, Auswahl, Terminator) genauer erklären.

5. Zusammengesetzte Daten

In diesem Kapitel befassen wir uns mit denjenigen ELAN-Sprachkonstrukten, die uns das Zusammenfassen mehrerer Datenobjekte unter einem Namen erlauben. Es wird dabei zwischen der Zusammenfassung gleichartiger (Reihungen) und ungleichartiger Datentypen (Strukturen) unterschieden. Neben der Zusammenfassung benötigen wir die Möglichkeit, auf die einzelnen Komponenten eines solchen Datenverbundes zuzugreifen. Diesen Zugriff nennt man Subskription bzw. Selektion.

5.1. Die Zusammenfassung gleichartiger Datentypen

Haben mehrere Datenobjekte den gleichen Datentyp, so können sie zu einem neuen Datenobjekt, einer Reihung ("row"), zusammengefaßt werden. Dies erfolgt durch eine Deklaration, in der

- das Schlüsselwort ROW
- die Anzahl der zusammengefaßten Komponenten der Reihung,
- der Datentyp der Komponenten,
- das Accessrecht der Zusammenfassung, sowie
- der Name der Reihung

angegeben wird.

Beispiel:

ROW 100 INT VAR feld

Diese Deklaration vereinbart eine Zusammenfassung von 100 Datenobjekten des Typs INT mit dem Namen "feld". Die gesamte Reihung hat das Accessrecht VAR. Wichtig bei dieser Deklaration ist, daß

ROW 100 INT

einen Datentyp vereinbart, der unterschiedlich zum Datentyp INT ist. So vereinbart etwa eine Deklaration, die zum Beispiel

ROW 101 INT

enthält, einen anderen, unterschiedlichen Datentyp. Dies bedeutet, daß - im Unterschied zu vielen anderen Programmiersprachen - die Anzahl der Komponenten (der Denoter hinter dem Schlüsselwort ROW) einer Reihung somit zum Datentyp gehört.

Die Angabe der "Komponenten-Anzahl" ist ein Denoter vom Datentyp INT (oder dessen Abkürzung; siehe LET-Vereinbarung Kap. 7). Der Denoter 0 (null) ist hierbei nicht erlaubt, da eine Reihung mit 0 (null) Elementen sinnlos ist. Da die Anzahl der Komponenten in den Datentyp der Reihung eingeht, muß sie vor der Laufzeit des Programmes festliegen. Folglich sind "dynamische Reihungen", d.h. Festlegung der Komponenten-Anzahl während der Laufzeit, nicht möglich.

5.1.1. Zugriff auf die Komponenten einer Reihung (Subskription)

Eine wichtige Konsequenz der Tatsache, daß jede Reihung einen eigenen Datentyp definiert, ist, daß keine Operationen auf Reihungen als Ganzes definiert sind, da offensichtlich beliebig viele Reihungstypen existieren und man daher auch beliebig viele dazugehörige Operatoren zur Verfügung stellen müßte.

Eine Ausnahme bildet der Zuweisungsoperator, der auch für Reihungen beliebigen Datentyps vorhanden ist, sofern die Zuweisung für die einzelnen Komponenten definiert ist. Das bedeutet, daß bei Reihungen die Zuweisung komponentenweise erfolgt. Selbstverständlich können ELAN-Übersetzer in diesen Fällen Optimierungen vornehmen. Wir werden später noch darstellen, wie es in ELAN möglich ist, weitere Operationen auf Reihungen in ihrer Gesamtheit zu definieren.

Gelingt es uns aber, einzelne Komponenten einer Reihung "herauszulösen", also auf die einzelnen Bestandteile einer Reihung zuzugreifen, dann können wir die Operationen anwenden, die für diese Komponenten definiert sind. Der Zugriff auf Komponenten einer Reihung wird in ELAN Subskription (in manchen Programmiersprachen Indizierung) genannt. Die Subskription erfolgt durch die Angabe des Reihungsnamens, der von einer in eckigen Klammern eingeschlossenen Formel gefolgt wird. Dieser Ausdruck wird Index genannt. Eckige Klammern werden hier verwendet, um die Subskription gegenüber den runden Klammern, die in Formeln auftreten können, abzuheben.

Beispiel:

```
ROW 100 INT VAR reihe;
reihe [1] := 1;
reihe [reihe [1] + 1] := 2;
reihe [3] := reihe [1] + reihe [2]
```

Dabei ist zu beachten, daß eine Subskription nicht das Accessrecht verändert. Das bedeutet, daß das Accessrecht einer Reihung auch für deren subskribierte Komponenten gilt.

5.1.2. Reihungs-Denoter ("row display")

Wie können wir Reihungen initialisieren? Eine - wenn auch etwas schreibaufwendige Lösung - haben wir im obigen Beispiel kennengelernt, bei der jede Komponente durch Subskription herausgelöst und ein Wert zugewiesen wurde. Jedoch ist es einfacher und für CONST-Datenobjekte geradezu notwendig, daß uns ein Reihungs-Denoter zur Verfügung steht, der die gleiche Rolle wie etwa ein INT-Denoter für einen INT-Datentyp spielt. Reihungsdenoter stehen uns in ELAN in Form des "row display" zur Verfügung. Ein "row display" besteht aus einer in eckigen Klammern eingeschlossenen Liste von Formeln. Jede Formel muß einen Datentyp liefern, der dem Komponenten-Datentyp der Reihung entsprechen muß.

Beispiel:

ROW 3 BOOL VAR tabelle :: [TRUE,TRUE,FALSE];

aber auch

ROW 4 INT VAR feld 4;
INT CONST eins :: 1, zwei :: 2, drei :: 3;
*feld 4 := [eins, eins*zwei, 2*drei, 4]*

Ein "row display" hat - ebenso wie andere Denoter - immer das Accessrecht CONST.

5.1.3. Reihungen von Reihungen

Wie oben erklärt, können von beliebigen Datentypen Reihungen gebildet werden. Da eine Reihung einen eigenen Datentyp definiert, ist es in ELAN erlaubt, Reihungen von Reihungen zu vereinbaren.

Beispiel:

ROW 3 ROW 4 REAL VAR matrix.

Dieses Datenobjekt hat

- den Datentyp ROW 3 ROW 4 REAL und
- den Namen "matrix" mit dem Accessrecht VAR.

Der Datentyp beschreibt eine Reihung von drei Komponenten des Datentyps ROW 4 REAL. Dieser wiederum besteht aus einer Reihung mit vier Komponenten vom Datentyp REAL.

Für Reihungen von Reihungen (mehrfache Reihungen) gilt folgendes:

- Von jedem ELAN-Datentyp dürfen Reihungen gebildet werden, also auch von Reihungen (Reihungen von Reihungen ...).
- Für mehrfache Reihungen sind - wie für einfache Reihungen - keine Operationen definiert, mit der Ausnahme der Zuweisung. Dabei ist zu beachten, daß

  ROW 4 ROW 5 REAL

  und

  ROW 5 ROW 4 REAL

  unterschiedlich im Datentyp sind, obwohl es sich um die gleiche Anzahl von Komponenten des Typs REAL handelt.
- Das Accessrecht gilt für alle möglichen Komponenten.
- Subskription und Denotierung sind möglich (vergl. nächsten Abschnitt).

5.1.4. Subskription und "row display" von mehrfachen Reihungen

Durch eine Subskription auf eine Reihung kann man eine Komponente einer Reihung ansprechen. Ist diese Komponente wiederum eine Reihung, so kann man auf deren Komponente wiederum durch eine Subskription gelangen.

Beispiel:

ROW 3 ROW 4 INT VAR alpha

Mit "alpha [2]" bekommt man das zweite ROW 4 INT-Element, während "alpha [2] [1]" eine INT-Komponente liefert.

Man kann also sagen, daß die Subskription eine Reihung von einem Datentyp "abschält", um auf die darunter liegende Komponente zu gelangen.

Natürlich muß es nun möglich sein, Denoter von mehrfachen Reihungen zu schreiben. Für die obige mehrfache Reihung könnte ein Denoter etwa folgendermaßen aussehen:

[[1,2,3,4], [1,2,3,4], [1,2,3,4]]

5.2. Die Zusammenfassung ungleichartiger Datentypen

Haben mehrere Datenobjekte einen unterschiedlichen Datentyp, so können die Objekte zu einer Struktur ("STRUCT") zusammengefaßt werden. Dies erfolgt ebenso wie bei Reihungen durch eine Deklaration, in der die unterschiedlichen Datenobjekte geklammert und mit dem vorangehenden "STRUCT"-Schlüsselwort sowie dem Namen mit dessen Accessrecht angegeben werden.

Beispiel:

STRUCT (TEXT name, INT geburtsjahr) VAR person

Diese Deklaration vereinbart die Zusammenfassung von einem TEXT- und einem INT-Datenobjekt, welches unter dem Namen "person" mit dessen Accessrecht VAR ansprechbar ist. Wichtig bei dieser Deklaration ist, daß

- die Namen der Komponenten (in der Beschreibung "fields", also Felder genannt) nicht mit einem Accessrecht versehen werden dürfen, da das Accessrecht für den Namen der gesamten Struktur gilt,
- die Felder durch Kommata voneinander getrennt werden,
- ein STRUCT einen eigenen Datentyp definiert; in diesem Fall gehen jedoch die Namen der Felder mit in den Datentyp ein; das bedeutet, daß etwa eine Deklaration wie

  STRUCT (TEXT name, INT alter) VAR person 2

  einen anderen Datentyp als der obige Strukturtyp darstellt, weil verschiedene Felder-Namen ("geburtsjahr" und "alter") verwendet wurden, obwohl die Feinstruktur, d.h. der Datentyp der Komponenten gleich ist.

5.2.1. Zugriff auf Komponenten von Strukturen (Selektion)

Wie bei Reihungen sind keine Operationen für Strukturtypen definiert. Eine Ausnahme bildet wiederum (vergl.5.1.1.) der Zuweisungsoperator, der auch für Strukturen beliebigen Datentyps vorhanden ist, sofern die Zuweisung für die einzelnen Komponenten existiert.

Im allgemeinen muß man daher auf die Feinstruktur, d.h. die Komponenten eines Strukturtyps zugreifen, um Operationen durchzuführen. Der Zugriff auf die Feinstruktur eines Strukturtyps wird Selektion, der Name eines Feldes Selektor genannt. Die Selektion erfolgt durch die Angabe

- des Namens der Struktur,
- eines Punktes und
- des Names der Komponenten der Struktur, die selektiert werden soll.

Beispiel:

```
STRUCT (TEXT name, INT alter, REAL gehalt) VAR personalangabe;
personalangabe.name:= "Rainer Hahn";
personalangabe.alter:= 35;
personalangabe.gehalt:= 10000.0
```

Die Selektion verändert nicht das Accessrecht einer Struktur (wie bei Reihungen).

5.2.2. Denoter für Strukturtypen (Konstruktor)

Selbstverständlich muß es die Möglichkeit geben, Denoter für Strukturtypen zu formulieren. Ein solcher Denoter wird in ELAN Konstruktor genannt. Er wird durch die Angabe des Strukturtyps (mit der Nennung der Komponenten) und die Angabe der geklammerten Felder denotiert.

Beispiel:

```
STRUCT (TEXT name, INT alter) VAR person;
person := STRUCT (TEXT name, INT alter) : ("hahn",35);
```

Wichtig ist, daß

- vor der geklammerten Angabe der Feinstruktur ein Doppelpunkt steht,
- der Typ (inklusive der Felder(-namen)) vollständig angegeben werden muß,
- die Reihenfolge der Komponenten bzw. der Felder einander entspricht.

Diese Form der Denotierung ist etwas schreibaufwendig. Wir werden jedoch in späteren Kapiteln das LET-Sprachkonstrukt vorstellen, das uns eine einfachere Formulierung von Konstruktoren erlaubt. Wichtig werden Konstruktoren auch für neudefinierte Datentypen. Konstruktoren haben, wie alle Denoter, das Accessrecht CONST.

5.2.3. Strukturen von Strukturen

Selbstverständlich kann man auch Strukturen von Strukturen bilden.

Beispiel:

```
STRUCT (STRUCT (TEXT name, REAL gehalt, INT alter) frau, mann,
        STRUCT (BOOL maennlich, INT alter) kind) VAR familie
```

Ein Zugriff auf ein Feld solch einer geschachtelten Struktur kann folgendermaßen geschrieben werden:

```
familie.kind.alter := 3
```

Mit "familie.kind" erhält man eine Struktur vom Typ

```
STRUCT (BOOL maennlich, INT alter)
```

Durch die Angabe des Feldnamens "alter" greift man auf eine Komponente dieses Datenobjekts zu. Die Konstruktion von Denotern für die Zuweisung und Initialisierung erfolgt nach dem gleichen Prinzip:

```
familie :=
STRUCT (STRUCT (TEXT name, REAL gehalt, INT alter) frau, mann,
        STRUCT (BOOL maennlich, INT alter) kind):

(STRUCT (TEXT name, REAL gehalt, INT alter):("Erna Meier",1700.0,33),
 STRUCT (TEXT name, REAL gehalt, INT alter):("Theo Meier",1620.0,31),
 STRUCT (BOOL maennlich, INT alter): (TRUE, 13))
```

Wie oben schon erwähnt, können wir die Formulierung von dem obigen Beispiel kürzer und übersichtlicher gestalten, wenn wir das LET-Sprachkonstrukt verwenden, das wir in Kapitel 7 kennenlernen werden.

5.3. Die Kombination von Reihungen und Strukturen

In ELAN ist es möglich, Reihungen und Strukturen zu verbinden. In diesem Abschnitt wollen wir auf einige mögliche "Mischformen" eingehen und in diesen Fällen die Selektion bzw. Subskription erklären.

5.3.1. Reihungen innerhalb von Strukturen

Es dürfen Reihungen innerhalb von Strukturen auftreten.

Beispiel:

```
STRUCT (ROW 5 TEXT vorname,
        TEXT nachname,
        ROW 5 INT alter) VAR familie
```

Der Zugriff erfolgt über zwei Stufen:

-"Herabsteigen" auf die Feinstruktur durch Selektion, etwa:

familie.vorname

liefert ein Objekt vom Datentyp ROW 5 TEXT

- und anschließende Subskription

familie.vorname [*3*]

liefert ein Objekt vom Datentyp TEXT.

Denoter werden durch den Konstruktor in Verbindung mit den "row display" gebildet, etwa:

```
STRUCT (ROW 5 TEXT vorname, TEXT nachname,
        ROW 5 INT alter):
        ([ "ingo","eva","adam","erna","brigitte"],
          "mueller",
         [32,27,2,3,5])
```

5.3.2. Reihungen von Strukturen

In ELAN sind Reihungen von Strukturen erlaubt, da es möglich ist, von jedem Datentyp eine Reihung zu bilden. Der Einfachheit wegen nehmen wir im folgenden Beispiel eine Reihung, die eine Struktur mit nur einem Feld enthält:

ROW 3 STRUCT (TEXT name) VAR familie

Der Zugriff auf ein Element erfolgt auch hier über zwei Stufen:

- Erst die Subskription, die uns aus der Reihung "familie" einen Strukturtyp liefert, etwa

  familie [*3*]

- anschließend Selektion, die ein Feldelement vom Typ TEXT liefert, etwa

  familie [*3*].*name*

Denoter werden nach den Regeln für Konstruktoren und Displays gebildet, z.B.

```
[STRUCT (TEXT name) : ("meier"),
 STRUCT (TEXT name) : ("mueller"),
 STRUCT (TEXT name) : ("schulze")]
```

5.3.3. Reihungen von Strukturen, die Reihungen enthalten

Die in den beiden vorherigen Abschnitten erklärten "Mischformen" darf man auch zusammen verwenden.

Beispiel:

ROW 2 STRUCT (ROW 3 TEXT name) VAR familie

Der Zugriff erfolgt in diesem Fall über drei Stufen:

- Zuerst die Subskription, die einen Datentyp STRUCT (ROW 3 TEXT name) liefert, etwa

  familie [*1*]

- Dann eine Selektion, das ein Feld des Strukturtyps liefert, in diesem Fall eine komplette Reihung:

  familie [*1*].*name*

- Darauf wiederum eine Subskription, die auf die Reihung innerhalb des Strukturtyps angewendet wird, um einen Datentyp TEXT zu liefern:

  familie [*1*].*name* [*3*]

Denoter werden gebildet, indem man die Regeln für Konstruktoren und Displays verwendet:

[*STRUCT (ROW 3 TEXT name) :* ([*"Hommel", "Jaehnichen", "Koch"*]),
 STRUCT (ROW 3 TEXT name) : ([*"Bartling", "Hahn", "Liedtke"*])]

Der Phantasie zur Bildung weiterer "Mischformen" von Reihungen und Strukturen sind keine Grenzen gesetzt. Dabei muß - wie wir hier gezeigt haben - die in der Deklaration festgelegte Reihenfolge von Strukturen und Reihungen bei der Selektion bzw. Subskription sowie bei Denotern eingehalten werden.

6. Zusammengesetzte Operationen

Nachdem wir im vorhergehenden Kapitel zusammengesetzte Datenobjekte besprochen haben, wollen wir jetzt zusammengesetzte Operationen behandeln. Einfache Operationen haben wir für die einfachen Datentypen und damit für die Komponenten von Reihungen und Strukturen schon kennengelernt. Nun wollen wir mehrere dieser Sprachkonstrukte hintereinander, bedingt oder wiederholt ausführen.

6.1. Einheiten ("units")

Einheiten sind neben Vereinbarungen (Deklarationen) die Grundbestandteile der Sprache ELAN. Zum Beispiel ist eine Formel, die eine Zuweisung ("assignation") enthält, eine Einheit. Es werden zwei Arten von Einheiten unterschieden:

- Werte liefernde Einheiten und
- Einheiten, die keine Werte liefern.

Letztere nennen wir in diesem Buch "Anweisungen", in Anlehnung an andere Programmiersprachen.

Einheiten sind:

- Primärausdrücke ("primary"); dazu gehören Bezeichner von Datenobjekten, Bezeichner subskribierter und selektierter Datenobjekte, Prozeduraufrufe (vergl. Kap. 10.), Refinement-Anwendungen (vergl. Kap. 8.), Ausdrücke (Formeln), Konstruktoren und "row displays" sowie Denoter,
- Abfragen,
- Auswahl,
- Wiederholungen.

Wir befassen uns zunächst mit denjenigen Einheiten, die keine Werte liefern. Im Kapitel 8 besprechen wir u.a. die Konstrukte, die verwendet werden müssen, um Einheiten in Werte liefernde Einheiten umzuwandeln. Besonders wichtig sind Einheiten, die den Ablauf von Programmen steuern. Diese Einheiten heißen auch Kontrollanweisungen (oder Kontrollstrukturen bzw. Steuerkonstrukte).

6.2. Die Aneinanderreihung von Einheiten (Sequenz)

Anweisungen bzw. Einheiten werden durch das "go on Symbol" getrennt, das durch das ";"-Zeichen repräsentiert wird. Beachte, daß das "go on Symbol" die Bedeutung von

"führe die nächste Einheit aus"

hat. Deshalb darf normalerweise (im Gegensatz zu anderen Sprachen der ALGOL-Sprachfamilien, wo eine "EMPTY"-Anweisung existiert) hinter der letzten Anweisung kein Semikolon stehen.

Beispiel:

```
einheit 1;
einheit 2;
   :
   :
einheit n
```

Außerdem wird das Semikolon benutzt, um Deklarationen untereinander und diese von Einheiten zu trennen. Desgleichen gilt für mehrere aufeinanderfolgende Pakete (vergl. Kap. 11.).

Das bedeutet, daß das ";"-Zeichen Einheiten voneinander trennt ("statement separator") und nicht - wie in einigen anderen Programmiersprachen - Einheiten beendet ("statement terminator").

6.3. Abfragen ("choice clause")

Mit Abfragen kann man die bedingte Ausführung von Anweisungen steuern. Es werden zwei Arten der Abfrage unterschieden:

6.3.1. Die einseitige Abfrage

Häufig soll eine Einheit oder Sequenz nur dann ausgeführt werden, wenn eine gewisse Bedingung eintritt. Dies läßt sich durch die einseitige Abfrage steuern. Sie besteht aus:

- Dem Schlüsselwort IF,
- einer Bedingung ("condition"),
- dem Schlüsselwort THEN,
- einer oder mehreren Anweisungen (Sequenz),
- dem Schlüsselwort END IF.

Beispiel:

```
INT VAR a,b;
get (a);    (* Einlesen der Werte, *)
get (b);    (* vergl. 12.4.1.      *)
IF a > b THEN
   (* Vertauschung von a und b *)
   INT CONST zwischenwert :: a;
   a := b;
   b := zwischenwert
END IF;
put (a);    (* Ausgeben der Werte, *)
put (b);    (* vergl. 12.4.2.      *)
```

Die Bedingung ("condition") muß aus einem Ausdruck bestehen, der einen Wert vom Typ BOOL liefert. Es kann sich dabei um Ausdrücke handeln, die durch einen Vergleichs-Operator ("relational operator") verbunden werden, etwa

netto gehalt + steuern > 10000

oder es kann von vornherein ein Ausdruck sein, der nur aus booleschen Variablen und Operatoren besteht, etwa

(kann elan AND programmieren) OR kann algol 68

oder einer Mischung dieser beiden Möglichkeiten, etwa

(gehalt + steuern) > 10000 AND deutscher

Die Anweisung(en) hinter dem Schlüsselwort THEN - der sogenannte "THEN part" - darf wiederum aus bedingten Abfragen bestehen (oder aus anderen Kontrollstrukturen), so daß geschachtelte Abfragen möglich sind.

Beispiel:

```
:
:
IF  zeichen = "(" THEN
    lies naechstes zeichen;
    IF  zeichen = " " THEN
        behandle kommentar
    END IF
END IF
:
:
```

Mit Refinements (vergl. Kap. 8) können solche geschachtelten Konstruktionen besser formuliert werden.

6.3.2. Die zweiseitige Abfrage

Will man zwei alternative Einheiten oder Sequenzen in Abhängigkeit von einer Bedingung durchführen, so kann man hierfür die zweiseitige Abfrage benutzen. Sie unterscheidet sich von der einseitigen Abfrage dadurch, daß dem THEN-Teil ein ELSE-Teil folgt.

Beispiel:

```
IF  wert gefunden
    THEN drucke wert
    ELSE erhoehe index
END IF
```

Im ELSE-Teil dürfen wiederum beliebige Einheiten oder Sequenzen stehen, also auch andere Kontrollanweisungen.

6.3.3. Die Abfragekette

Will man Abfrageketten konstruieren, d.h. in einem ELSE-Teil wieder eine bedingte Anweisung schreiben, so ergibt sich der ärgerliche Umstand, daß mehrere END IFs geschrieben werden müssen.

Beispiel:

```
INT VAR a,b,c;
get (a);
get (b);
get (c);
IF a > b AND a > c
   THEN put (a)
   ELSE
   IF b > c
      THEN put (b)
      ELSE put (c)
   END IF
END IF
```

Um dies zu vermeiden, bietet ELAN das ELIF-Sprachelement (eine Zusammensetzung der Worte ELSE und IF).

Beispiel:

```
INT VAR a,b,c;
get (a);
get (b);
get (c);
IF a > b AND a > c
   THEN put (a)
ELIF b > c
   THEN put (b)
   ELSE put (c)
END IF
```

Man beachte, daß hier nur ein END IF geschrieben werden darf.

6.4. Die Auswahl ("multiple choice clause")

Es gibt Anwendungen, in denen mehrere alternative Einheiten oder Sequenzen in Abhängigkeit von den Werten eines Datenobjekts durchgeführt werden sollen, im Gegensatz zum ELIF-Sprachelement, bei dem mehrere Bedingungen notwendig sind.

Beispiel:

```
:
:
SELECT monat OF
   CASE 2: IF schaltjahr
              THEN tage := 29
              ELSE tage := 28
           END IF
   CASE 4,6,9,11: tage := 30
   CASE 1,3,5,7,8,10,12: tage := 31
   OTHERWISE kein monat
END SELECT;
:
:
```

Die Auswahl besteht aus mehreren Teilen:

- Zwischen den Schlüsselworten SELECT und OF steht die Fall-Auswahl ("case selection"). Diese muß aus einer Formel ("formula") bestehen, die ein INT-Datenobjekt liefert. Der INT-Wert dieses Datenobjekts wird benötigt, um einen der folgenden "case parts" auszuwählen.
- Die Fälle ("case parts") bestehen aus dem Schlüsselwort CASE, gefolgt von mindestens einem INT-Denoter. Es darf aber auch eine Liste von INT-Denotern stehen, die nicht wertemäßig geordnet sein muß. Diese Denoter werden benötigt, um als Fallunterscheidungsmerkmal für die "case selection" zu dienen. Die Liste der INT-Denoter wird durch ein ":"-Zeichen abgeschlossen. Danach dürfen eine oder mehrere Einheiten stehen (Sequenz), die ausgeführt werden, sofern die Formel hinter SELECT den Wert eines aufgeführten INT-Denoters dieses "case parts" ergibt. Die Einheiten für einen "case part" werden durch das nächste Schlüsselwort begrenzt (ein anderes CASE, OTHERWISE oder END SELECT).

  Anstatt eines INT-Denoter darf auch ein mit dem LET-Konstrukt (vergl. 7) vereinbarter Name erscheinen, der einen INT-Denoter bezeichnet.
- Der optionale "otherwise part" muß als letzte Anweisung der Auswahl stehen. Er wird für diejenigen Fälle benötigt, in denen die Auswertung des INT-Ausdrucks hinter dem Schlüsselwort SELECT kein entsprechender "case part" findet. Obwohl der "otherwise part" optional ist, sollte er immer verwendet werden, da Fehlerfälle auf diese Weise abgefangen werden können.
- Beendet wird die Auswahl - wie bei allen Kontrollanweisungen - durch das Schlüsselwort END und (in diesem Fall) SELECT.

6.5. Wiederholungen ("repetitions")

Das Sprachelement der Wiederholung dient zur mehrfachen Ausführung einer oder mehrerer Einheiten. ELAN bietet zu diesem Zweck verschiedene Konstrukte, die miteinander kombiniert werden können. Die sprachliche Konstruktion der Wiederholung wird u.a. auch "Schleife" genannt. Die Einheiten, die mehrfach ausgeführt werden sollen, d.h. die sich innerhalb der Schleife befinden, werden Schleifenrumpf genannt. Die Bedingung, die die Anzahl der Ausführungen der Schleife kontrolliert, wird Abbruchbedingung genannt. Im Schleifenrumpf müssen Einheiten enthalten sein, die den Wert der Bedingung nach endlich vielen Durchläufen so verändern, daß die Schleife abgebrochen wird.

6.5.1. Die Endlosschleife

Die Endlosschleife enthält keine Abbruchbedingung. Die zu wiederholenden Einheiten (Schleifenrumpf) werden von den Schlüsselworten REP (bzw. REPEAT) und END REP (bzw. END REPEAT) eingeschlossen.

Beispiel:

```
REP
     lies eingabewerte;
     berechnung;
     drucke ausgabewerte
END REP
```

Selbstverständlich kann man nur theoretisch eine Endlosschleife schreiben, d.h. ein Programm, das "ewig" läuft. Tatsächlich werden Programme immer begrenzt sein.

Die Endlosschleife ist von geringerer Bedeutung. Sie wird hier der Vollständigkeit halber aufgeführt, weil sie als Basiskonstrukt für alle anderen Schleifenformen verwendet wird.

Die Endlosschleife kann in Verbindung mit einem Terminator (s. 8.5.) eine sinnvolle Anwendung finden.

6.5.2. Die abweisende Schleife (WHILE-Konstrukt)

Eine abweisende Schleife erhält man, indem vor das Schlüsselwort REP ein optionaler "while part" geschrieben wird. Der "while part" bestimmt wie oft bzw. ob der Schleifenrumpf ausgeführt wird.

Beispiel:

```
WHILE noch eingabewert vorhanden REP
:
:
END REP
```

Hinter dem Schlüsselwort WHILE steht eine Bedingung ("condition"), d.h. ein Ausdruck, der einen booleschen Wert liefern muß. Solange ("WHILE") seine Auswertung TRUE ergibt, wird der Schleifenrumpf ausgeführt. Daher wird die Bedingung bei jedem neuen Schleifendurchlauf neu ausgewertet.

Ergibt die Bedingung bei der ersten Auswertung (d.h. vor der ersten Ausführung des Schleifenrumpfes) den Wert FALSE, so wird der Schleifenrumpf übergangen, und es wird diejenige Einheit ausgeführt, die den Schlüsselworten END REP folgt (abweisende Schleife). Oft wird die WHILE-Schleife auch zum Zählen verwendet, obwohl in ELAN ein besonderes Sprachkonstrukt (die Zählschleife) für diesen Zweck existiert.

Beispiel:

```
INT VAR zaehler :: 1;
ROW 100 INT VAR feld;
WHILE zaehler <= 100 REP
  get (feld [zaehler]);
  zaehler := zaehler + 1          (* oder : zaehler INCR 1 *)
END REP
```

6.5.3. Die nichtabweisende Schleife (UNTIL-Konstrukt)

Bei der nichtabweisenden Schleife wird der Schleifenrumpf auf jeden Fall einmal ausgeführt, weil die Abbruchbedingung erst hinter dem Schleifenrumpf steht (also direkt vor dem Schlüsselwort END REP) und auch dort ausgewertet wird.

Das gleiche Beispiel wie bei der WHILE-Schleife kann mit der UNTIL-Schleife folgendermaßen formuliert werden:

```
INT VAR zaehler :: 1;
ROW 100 INT VAR feld;
REP
    get (feld  [zaehler]);
    zaehler := zaehler + 1
UNTIL zaehler > 100 END REP
```

Hinter dem Schlüsselwort UNTIL steht ein Bedingung, die einen booleschen Wert liefern muß. Die Schleife wird solange ausgeführt, bis ("UNTIL") die Abbruchbedingung den Wert TRUE liefert (beachte den Unterschied zur WHILE-Abbruchbedingung).

Die UNTIL-Bedingung wird - ebenso wie die WHILE-Bedingung - bei jeder Ausführung der Schleife erneut ausgewertet und überprüft.

Dabei ist ebenso wie bei dem WHILE-Konstrukt wichtig, daß innerhalb des Schleifenrumpfes Einheiten enthalten sind, die den Wert der Bedingung so verändern, daß ein Abbruch der Schleife möglich wird.

6.5.4. Die Zählschleife (FOR-Konstrukt)

Die Zählschleife wird dann verwendet, wenn der Schleifenrumpf mit aufeinanderfolgenden Werten (Abstand 1) durchlaufen werden soll.

Beispiel:

```
INT VAR zaehler;
ROW 100 INT VAR feld;
FOR zaehler FROM 1 UPTO 100 REP
    get (feld [zaehler])
END REP
```

Der "for part" wird vor das Schlüsselwort REP geschrieben. Er besteht aus:

- dem Schlüsselwort FOR, gefolgt von einem Bezeichner (der "Laufvariablen"), der mit dem Datentyp INT und dem Accessrecht VAR deklariert worden ist; er darf keine subskribierte oder selektierte Variable sein und darf im Schleifenrumpf nicht verändert werden;
- dem Schlüsselwort FROM, gefolgt von einer Formel, die ein INT-Datenobjekt liefert, dessen Wert den Anfangswert der Laufvariablen festlegt;
- entweder dem Schlüsselwort UPTO (zählt mit 1 aufwärts) oder DOWNTO (zählt mit 1 abwärts); diese Schlüsselworte geben die Richtung ("direction") an, in der die Zählschleife arbeiten soll; dem Schlüsselwort UPTO bzw. DOWNTO folgt eine Formel, die ein INT-Datenobjekt liefert, dessen Wert den Endwert der Zählschleife angibt; wird dieser Endwert überschritten, so wird die Ausführung der Schleife abgebrochen; in diesem Fall ist der Wert der Laufvariablen (nach Verlassen der Schleife) nicht definiert; wird hingegen die Schleife vor dem Überschreiten des Endwertes abgebrochen (z.B. über einen Terminator, eine WHILE- oder UNTIL-Bedingung), so hat die Laufvariable nach dem Verlassen den aktuellen Wert.

Die Formeln hinter den Schlüsselworten FROM (Anfangswert) und TO (Endwert) werden - im Gegensatz zur WHILE- und UNTIL-Schleife - nur einmal am Anfang der Schleifen-Elaboration ausgewertet.

Sollen FOR und WHILE gemeinsam verwendet werden, so müssen sie in dieser Reihenfolge stehen.

Beispiel:

```
FOR index FROM 1 UPTO anzahl der elemente
WHILE element noch nicht gefunden REP
:
:
END REP
```

6.6. Dynamische Initialisierung

Deklarationen von Datenobjekten dürfen (fast) an jede Stelle eines Programms geschrieben werden, also auch innerhalb von Kontrollstrukturen. Das folgende Beispiel, das innerhalb einer bedingten Anweisung eine Deklaration mit Initialisierung verwendet, um die Werte zweier Datenobjekte ("a" und "b") zu vertauschen, soll dies verdeutlichen.

Beispiel:

```
    :
    :
IF ... THEN
       INT CONST hilf :: a;
       a := b;
       b := hilf
       ELSE
       :
       :
END IF
```

Dabei ist folgendes zu beachten:

- Die Deklaration des Datenobjektes "hilf" wird nur einmal durchgeführt, da weitere Deklarationen, die Datenobjekte ebenfalls mit dem Namen "hilf" vereinbaren ungültig (fehlerhaft) wären, da dann mehrere Objekte mit gleichen Namen existieren würden.
- Die Initialisierung wird jedoch so oft durchgeführt, wie sie durchlaufen wird. Daher kann man sich eine Deklaration mit Initialisierung als zwei Anweisungen vorstellen:

  a) Deklaration des Datenobjekts (einmalig) und

  b) Initialisierung, die auf eine Zuweisung zurückgeführt wird.

Besonders deutlich wird die dynamische Initialisierung (d.h. eine Initialisierung, die unter Umständen mehrfach durchgeführt wird) bei Schleifen.

Beispiel:

```
INT VAR index;
ROW 100 INT VAR feld;
lies feldkomponenten ein;
FOR index FROM 2 UPTO 100 REP
  INT VAR zeiger darauf :: index - 1;
  INT VAR max element :: feld [zeiger darauf];
  durchsuche feld nach groesstem element;
  vertausche max element mit feld (* [index - 1] *)
END REP
```

Bei jedem Durchlauf der Schleife werden die Datenobjekte "max element" und "zeiger darauf" neu initialisiert, jedoch existiert jeweils nur ein Datenobjekt dieses Namens.

7. Die schwache Datenabstraktion (LET-Konstrukt)

Als schwache Datenabstraktion werden in ELAN die Abkürzungsvereinbarungen ("shorthand declaration") bezeichnet. Sie werden in den Programmtext eingeführt, um Denotern von einfachen Datentypen oder Datentypen (insbesondere: Reihungen und Strukturen) einen Namen zu geben. Das bedeutet, daß durch die schwache Datenabstraktion ein neuer Name eingeführt wird, jedoch die Eigenschaften des Objekts beibehalten werden.

Mit der schwachen Datenabstraktion kann man zwei Ziele erreichen:

- Dokumentation. Durch die Namensgebung kann man eine unmittelbare Dokumentation im Programmtext erreichen.
- Leichte Änderbarkeit. Durch die Namensgebung kann ein Programmtext leicht geändert werden, da nur an einer Stelle im Programmtext eine Veränderung vorgenommen werden muß. Bei erneuter Übersetzung des Programms wird automatisch an allen Anwendungsstellen der Abkürzung der neue Text eingesetzt.

Dabei ist zu beachten, daß die in den Abkürzungen benutzten Namen nur in dem entsprechenden Gültigkeitsbereich dieser Namen verwendet werden können (vergl. Kap. 11.).

7.1. Namen für Denoter

Mit Hilfe des LET-Konstrukts können Namen für Denoter der einfachen Datentypen (INT, REAL, BOOL und TEXT) in ein Programm eingeführt werden. Diese Namen, die für Denoter stehen, können an jeder Stelle stehen an der auch die Denoter selbst verwendet werden können.

Beispiel:

```
LET anzahl personen = 100;
ROW anzahl personen INT VAR gehaltsliste;
INT VAR index;
FOR index FROM 1 UPTO anzahl personen REP
  :
  :
END REP;
  :
  :
```

Die Abkürzungsvereinbarung besteht aus

- dem Schlüsselwort LET,
- dem Namen für den abzukürzenden Denoter,
- dem Gleichheitszeichen und
- dem abzukürzenden Denoter.

Es darf auch eine Liste von Abkürzungen hinter dem Schlüsselwort LET geschrieben werden.

Beispiel:

```
LET anzahl elemente = 100,
    ueberschrift = "ELAN - Handbuch",
    schalter offen = TRUE
```

Abkürzungen werden von dem ELAN-Übersetzer textuell eingesetzt (ähnlich den Refinements, vergl. Kap.8).

7.2. Namen für Datentypen

Das LET-Konstrukt kann ebenfalls für die Namensgebung von Datentypen verwendet werden. Wie bei den Namen für Denoter können auch diese Namen an jeder Stelle stehen, an der auch der Datentyp selbst stehen kann. Diese Namensgebung ist besonders für Datentypen von Reihungen und Strukturen sinnvoll und recht praktisch (vergl. Kap. 5.).

Beispiel:

```
LET MATRIX = ROW 100 ROW 5 INT;
MATRIX VAR mat
```

Wie man diesem Beispiel entnehmen kann, muß hinter dem Schlüsselwort LET der Name des abzukürzenden Typs als Schlüsselwort (also in großen Buchstaben) geschrieben werden, da der Name als Abkürzung für einen Typ in Deklarationen verwendet wird. Zu beachten ist dabei, daß in dem LET-Konstrukt nur der Datentyp, d.h. kein Accessrecht angegeben werden darf.

Wie bei den Abkürzungen für Denoter kann eine Liste von Abkürzungen für Datentypen gebildet werden.

Beispiel:

```
LET TABELLE = ROW 100 INT,
    TEXTMATRIX = ROW 5 ROW 10 TEXT
```

7.3. Gemeinsame Verwendung von Namen für Denoter und Datentypen

Namen als Abkürzungen für Denoter und Datentypen können gemeinsam verwendet und sogar geschachtelt werden. Dadurch wird eine bequeme Handhabung dieser Elemente möglich.

Beispiel:

```
LET max = 100,
    ADRESSE = STRUCT (INT postleitzahl,
                      TEXT strasse,
                      INT hausnummer),
    PERSON = STRUCT  (TEXT name, vorname,
                      ADRESSE adr,
                      INT alter),
    MITGLIEDSLISTE = ROW max PERSON;

MITGLIEDSLISTE VAR elan fan club
(* Die Anzahl "max" der Mitglieder dieses ominoesen Clubs laesst sich durch
   eine Aenderung leicht modifizieren *)
```

8. Refinements: die schwache algorithmische Abstraktion

Sinngemäß zu den Abkürzungsvereinbarungen für Denoter und Datentypen lassen sich in ELAN auch Ausdrücke oder mehrere Einheiten mit einem Namen versehen. Entsprechend der schwachen Datenabstraktion wird dieser Vorgang schwache algorithmische Abstraktion genannt. Das Sprachelement, das diese Namensgebung ermöglicht, heißt in ELAN Refinement.

Refinements dienen mehreren Zielen:

- "top-down"-Programmierung. Das Refinement ist ein ideales Instrument für die Methode der "schrittweisen Verfeinerung".
- Übersicht. Durch die Verwendung von Refinements entsteht eine Baumstruktur des Programmtextes. Diese erleichtert die leichte
- Wartbarkeit und Fehlersuche.
- Dokumentation. Durch die Namensgebung von Teil-Algorithmen oder Ausdrükken wird eine unmittelbare Dokumentation erreicht.

8.1. Das Refinement als Instrument zur schrittweisen Verfeinerung

Das Prinzip der schrittweisen Verfeinerung mit Hilfe von Refinements wollen wir an den folgenden Lösungswegen darstellen. Zunächst geben wir dabei Teilalgorithmen an den Stellen Namen, an denen eigentlich diese Teilalgorithmen explizit stehen müßten. Die Aufführung eines solchen Namens heißt Refinement-Anwendung. Wenn wir den Algorithmus derart formuliert haben, so folgt die Konkretisierung der mit Namen versehenen Teilalgorithmen (Refinements). Diese nennen wir Refinement-"Definition".

Beispiel:

```
bubble sort.

bubble sort:
    LET anzahl elemente = 500;
    ROW anzahl elemente INT VAR liste;
    hole unsortierte liste;
    sortiere liste;
    drucke sortierte liste.

hole unsortierte liste:
    INT VAR elementnummer;
    FOR elementnummer FROM 1 UPTO anzahl elemente REP
        get (liste [elementnummer])
    END REP.

sortiere liste:
    INT VAR durchgangszaehler;
    FOR durchgangszaehler FROM 1 UPTO anzahl elemente - 1 REP
        BOOL VAR keine vertauschung erfolgt :: TRUE;
        vertausche wenn notwendig in unsortierter restliste
    UNTIL keine vertauschung erfolgt END REP.

vertausche wenn notwendig in unsortierter restliste:
    FOR elementnummer FROM 1 UPTO anzahl elemente - durchgangszaehler REP
        vergleiche und vertausche elemente wenn notwendig
    END REP.

vergleiche und vertausche elemente wenn notwendig:
    IF liste [elementnummer] > liste [elementnummer + 1]
        THEN vertausche;
             keine vertauschung erfolgt := FALSE
    END IF.

vertausche:
    INT CONST tauscher :: liste [elementnummer];
    liste [elementnummer] := liste [elementnummer + 1];
    liste [elementnummer  + 1] := tauscher.

drucke sortierte liste:
    FOR elementnummer FROM 1 UPTO anzahl elemente REP
        put (liste [elementnummer])
    END REP
```

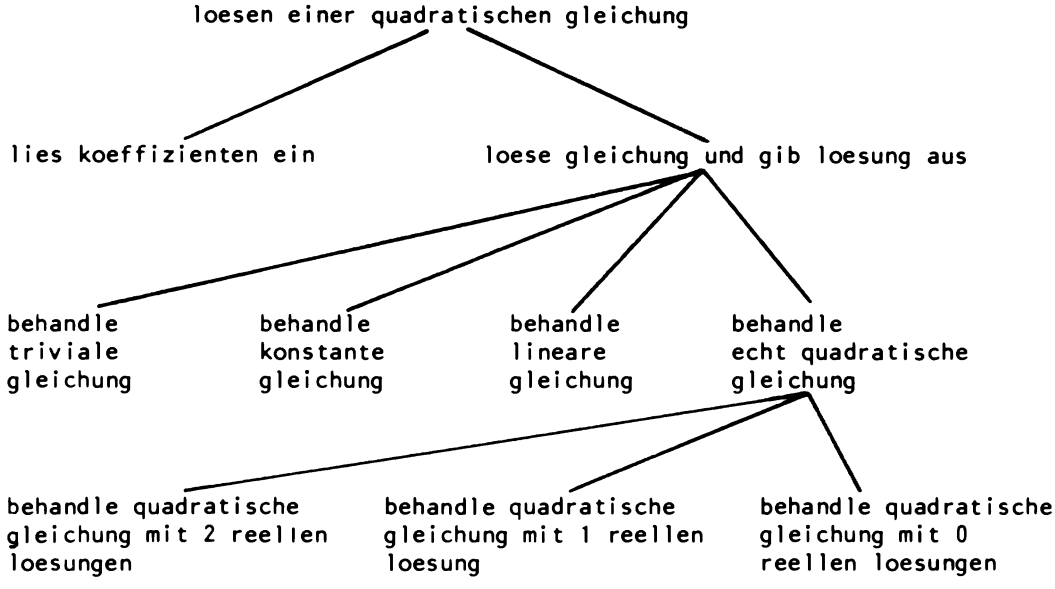
Da das Refinement ein Sprachelement von besonderer Wichtigkeit in ELAN ist, fügen wir ausnahmsweise gleich noch ein weiteres Beispiel an.

Beispiel:

```
loesen einer quadratischen gleichung.
     (* der Form: a * x ** 2 + b * x + c = 0.0 *)

loesen einer quadratischen gleichung:
     lies koeffizienten ein;
     loese gleichung und gib loesung aus.

lies koeffizienten ein:
     REAL VAR  a, b, c;
     get (a);
     get (b);
     get (c).

loese gleichung und gib loesung aus:
     IF    a = 0.0 AND b = 0.0 AND c = 0.0
           THEN behandle triviale gleichung
     ELIF  a = 0.0 AND b = 0.0
           THEN behandle konstante gleichung
     ELIF  a = 0.0
           THEN behandle lineare gleichung
           ELSE behandle echt quadratische gleichung
     END IF.

behandle triviale gleichung:
     put ("Loesung: alle reellen Zahlen").

behandle konstante gleichung:
     put ("Keine loesung, da die Gleichung lautet: 0.0 =");
     put (c).

behandle lineare gleichung:
     put ("Loesung:");
     put (- c/b);
     put ("  Die Gleichung war linear.").

behandle echt quadratische gleichung:
     REAL VAR diskriminante :: (b * b/(4.0 * a) - c)/a;
     IF    diskriminante > 0.0
           THEN behandle quadratische gleichung mit 2 reellen loesungen
     ELIF  diskriminante = 0.0
           THEN behandle quadratische gleichung mit 1 reellen loesung
           ELSE behandle quadratische gleichung mit 0 reellen loesungen
     END IF.

behandle quadratische gleichung mit 2 reellen loesungen:
     put ("1.Loesung:");
     put (-b/(2.0 * a) + sqrt (diskriminante));
                (* sqrt = squareroot, Quadratwurzel, Abschnitt 13.3 *)
     put ("2.Loesung:");
     put (-b/(2.0 * a) - sqrt (diskriminante)).

behandle quadratische gleichung mit 1 loesung:
     put ("Loesung:");
     put (-b/(2.0 * a));
     put ("  Die Loesungen fallen ueberein, da die Diskriminante 0 ist.").

behandle quadratische gleichung mit 0 loesungen:
     put ("Es existieren keine Loesungen im Reellen, da Diskriminante < 0.")
```

Die Vorgehensweise ist also folgende: ein (u.U. unübersichtliches) Problem wird solange in Teilprobleme zergliedert, bis sich diese leicht und übersichtlich lösen lassen.

Diese schrittweise Verfeinerung des Problems läßt sich graphisch in einem Baumdiagramm darstellen:



Die Vorteile eines solchen Vorgehens bei der Konstruktion eines Algorithmus liegen auf der Hand:

- Je nach der Ebene des Zerlegungsprozesses braucht man sich nur mit Teilaspekten eines Problems zu befassen.
- Durch die Benennung von Teilproblemen bei der Formulierung eines Algorithmus wird die Dokumentation gefördert.
- Die Verwendung von Refinements kann das Entwickeln von Algorithmen fördern, indem für Teilprobleme, deren Lösung noch nicht vorliegt, erst einmal Namen verwendet werden. Ist der Algorithmus vollständig mit Hilfe von Refinements formuliert, so liegt unmittelbar auch eine Dokumentation des Lösungsweges vor, sofern gute Namen gewählt wurden.
- Dadurch können Dritte einen Algorithmus besser verstehen. Somit lassen sich Wartungsarbeiten bzw. Modifikationen leichter durchführen.

8.2. Die Konstruktion von Refinements

Refinement-"Definitionen" werden, wie wir in den vorangegangenen Beispielen sehen, stets am Ende eines Algorithmus vorgenommen und durch einen Punkt vom Algorithmus und auch untereinander getrennt. Man kann auch sagen: sie werden von einem Punkt eingeleitet. Hinter dem letzten Refinement darf also kein Punkt stehen.

Bei der Verwendung von Refinements sind einige Regeln zu beachten:

a) Wir haben zu unterscheiden zwischen der Refinement-"Definition" und der Refinement-Anwendung. Eine Refinement-Anwendung besteht aus dem Namen des Refinements. Die "Definition" eines Refinements besteht aus der Angabe des Refinement-Namens, dem Doppelpunkt und einer oder mehreren Einheiten, die ihrerseits wiederum Refinement-Anwendungen enthalten dürfen.

b) Refinements werden textuell an der Stelle eingesetzt, an der Refinement-Anwendungen erscheinen. Das ist einer der Gründe, warum Refinements nicht direkt oder indirekt rekursiv angewendet werden dürfen.

c) In Refinements dürfen Datenobjekte deklariert werden. Refinements besitzen jedoch keine eigenen Datenräume. Das bedeutet, daß die in einer Refinement-"Definition" deklarierten Datenobjekte in der gesamten Umgebung gelten (vergl. dazu Kap. 11.2.2.).

d) Befinden sich in einer Refinement-"Definition" Deklarationen mit Initialisierungen, so werden diese Initialisierungen bei jeder Elaboration der entsprechenden Refinement-Anwendung durchgeführt.

   Beispiel (vergl. Beispiel "bubble sort" in Abschnitt 8.1.):

```
:
:
IF liste [elementnummer] > liste [elementnummer + 1]
   THEN vertausche;
        keine vertauschung erfolgt := FALSE
END IF.

vertausche:

INT CONST tauscher :: liste [elementnummer];
:
:
```

   Das Datenobjekt "tauscher" wird jedesmal initialisiert, wenn das Refinement "vertausche" in der Abfrage angewendet wird. Die Abfrage selbst ist wiederum ein Refinement, das innerhalb einer (faktisch geschachtelten) Schleife angewendet wird, die ihrerseits mehrfach durchlaufen wird.

e) Refinements können keine Parameter haben (vergl. dazu "Prozeduren" Kap. 10.), dürfen jedoch Werte liefern (vergl. 8.4.).

8.3. Werte liefernde Einheiten

Bestimmte ELAN-Sprachkonstrukte können durch ihre Elaboration Werte eines bestimmten Datentyps liefern.

Wir werden hier die bis jetzt behandelten Sprachkonstrukte besprechen, die Werte liefern können. Weitere Werte liefernde Konstrukte, wie das Refinement, der Terminator, Prozeduren und Operatoren werden wir in den nächsten Abschnitten bzw. Kapitel 9. behandeln.

a) Ein Bezeichner eines Datenobjekts, eine Subskription und eine Selektion liefern Werte vom selben Datentyp, mit dem sie vereinbart wurden (bzw. bei Subskription und Selektion den Datentyp der subskribierten bzw. selektierten Komponente). Das Accessrecht ist dasjenige Accessrecht, mit dem die Datenobjekte vereinbart wurden. Denoter liefern den Datentyp, der durch ihre Darstellung festgelegt wird. Ihr Accessrecht ist CONST.

b) Ausdrücke liefern, wie wir schon unter 4.3. gesehen haben, Werte. Der Typ des Ergebnisses wird durch die zuletzt ausgeführte Operation bestimmt. Die Auswahl dieser Operation ist u.a. vom Typ ihrer Operanden (bzw. Parameter) abhängig. Das Accessrecht des Ergebnisses ist im Gegensatz zu dem unter a) Aufgeführten immer CONST.

c) Sequenzen liefern einen Wert, sofern die letzte Einheit der Sequenz einen Wert liefert.

d) Abfragen ("choice clause") liefern Werte, wenn die jeweils letzte ausgeführte Einheit einen Wert liefert. Einseitige Abfragen können aus naheliegendem Grund keine Werte liefern: ist die Bedingung erfüllt, so wird der THEN-Teil abgearbeitet, liefert also einen Wert; ist die Bedingung nicht erfüllt, so wird der THEN-Teil übergangen, und die Abfrage liefert keinen Wert. Dies widerspricht aber der Forderung, daß entweder eine Einheit stets einen Wert liefert oder stets keinen. Ist hingegen auch ein ELSE-Teil vorhanden, so können diese zweiseitige Abfrage oder auch Abfrageketten mit ELSE-Teil Werte liefern.
Hierbei müssen die jeweilig letzten Einheiten aller THEN-Teile und des ELSE-Teils Werte liefern, die im Datentyp übereinstimmen müssen. Das Accessrecht dieser Werte bestimmt sich durch das Accessrecht der entsprechenden Einheiten. Sind sämtliche Accessrechte gleich, so hat die gesamte Abfrage(-kette) dieses Accessrecht. Treten verschiedene Accessrechte auf, so wird eine Balancierung durchgeführt, d.h. die Abfrage (-kette) erhält das Accessrecht CONST (vergl. 4.6.).

Beispiel (zweiseitige Abfrage):

```
REAL VAR gehalt, kindergeld;
INT VAR anzahl kinder;
:
:
IF anzahl kinder > 1
   THEN berechne kindergeld;
        gehalt + kindergeld
   ELSE gehalt
END IF
```

In diesem Beispiel wird ein Datenobjekt vom Datentyp REAL mit dem Accessrecht CONST geliefert. (Im ELSE-Teil wird eine Balancierung durchgeführt).

Beispiel (Abfragekette):

```
INT VAR a, b, c;
:
:
IF a > b AND a > c
   THEN a
ELIF b > c
   THEN b
   ELSE c
END IF
```

e) Die Auswahl ("multiple choice clause") liefert einen Wert, wenn die letzte Einheit eines jeden "case parts" und des "otherwise parts" einen Wert liefert. Dabei müssen die Datentypen der gelieferten Werte der "case parts" und des "otherwise parts" gleich sein. Liefern die einzelnen Teile verschiedene Accessrechte, so werden diese (auf CONST) balanciert.

Beispiel:

```
SELECT monat OF
   CASE 2 : IF schaltjahr
               THEN 29
               ELSE 28
            END IF
            (* Abfrage liefert einen INT - Wert *)
   CASE 4,6,9,11 : 30
   CASE 1,3,5,7,8,10,12, : 31
   OTHERWISE fehlermeldung; 0
END SELECT
```

Wie wir an diesem Beispiel sehen, ist es erlaubt, die in a) bis d) aufgezeigten Möglichkeiten miteinander zu kombinieren.

Manchmal ist es notwendig - obwohl nicht immer inhaltlich erforderlich - bei der Abfrage oder der Auswahl einer Alternative einen Wert anzugeben, damit das gesamte Konstrukt einen Wert liefert. Vergleiche dazu das vorangegangene Beispiel ("otherwise part"). Manche Konstrukte in ELAN liefern prinzipiell keine Werte. Dazu gehören u.a.

- die Schleife,
- die Einheiten, die "void" liefern, wie z.B. eine Druckanweisung,
- Deklarationen.

8.4. Werte liefernde Refinements

Nachdem wir einige bereits besprochene Einheiten kennengelernt haben, die Werte liefern können, wollen wir zeigen, wie auch Refinements Werte liefern. Dazu ist es notwendig, daß die letzte ausführbare Einheit eines Refinements einen Wert liefert. Der zu liefernde Wert eines Refinements (Typ und Accessrecht) entspricht dem der letzten Einheit.

Beispiel:

```
(* durchsuchen einer matrix *)
LET n = 50,
   MATRIX = ROW n ROW n INT;
MATRIX VAR mat;
   lies werte der matrix
   lies gesuchten wert;
IF wert gefunden
   THEN drucke index
END IF.

lies werte der matrix:
   (* nicht naeher beschrieben *)
   .
lies gesuchten wert:
INT VAR wert;
   get (wert).

wert gefunden:
BOOL VAR gefunden :: FALSE;
   durchsuche alle zeilen;
   gefunden.                (* Hier liefert das Refinement einen Wert. *)

durchsuche alle zeilen:
INT VAR zeile;
   FOR zeile FROM 1 UPTO n REP
       durchsuche alle spalten dieser zeile
   UNTIL gefunden END REP.

durchsuche alle spalten dieser zeile:
INT VAR spalte;
   FOR spalte FROM 1 UPTO n REP
       IF mat [zeile] [spalte] = wert
           THEN gefunden := TRUE
       END IF
   UNTIL gefunden END REP.

drucke index:
   put ("Gesuchter Index: Zeile =");
   put (zeile);
   put ("  Spalte =");
   put (spalte)                .
```

8.5. Terminatoren (LEAVE-Konstrukt)

Mit Hilfe eines Terminators kann die Abarbeitung eines Refinements (einer Prozedur oder eines Operators) vorzeitig beendet werden. Als Beispiel verwenden wir wieder das Durchsuchen einer Matrix nach einem gegebenen Wert:

```
LET n = 50,
    MATRIX = ROW n ROW n INT;
MATRIX VAR mat;
    lies werte der matrix;
    lies gesuchten wert;
    IF wert gefunden THEN drucke index END IF.
:
wert gefunden:
INT VAR zeile;
    FOR zeile FROM 1 UPTO n REP
        durchsuche alle spalten dieser zeile
    END REP;
    FALSE.

durchsuche alle spalten dieser zeile:
INT VAR spalte;
    FOR spalte FROM 1 UPTO n REP
       IF mat [zeile] [spalte] = wert
          THEN LEAVE wert gefunden WITH TRUE
       END IF
    END REP.
```

In diesem Beispiel wird mit dem Terminator (LEAVE-Konstrukt) das Refinement "wert gefunden" aus dem Refinement "durchsuche alle spalten dieser zeile" verlassen.

Der Terminator besteht aus

- dem Schlüsselwort LEAVE, gefolgt von dem Namen des Refinements, welches vorzeitig beendet werden soll (man kann auch Prozeduren und Operatoren mit dem LEAVE-Konstrukt beenden, vergl. Kap 10.). Dabei muß der Refinement-Name, der dem LEAVE folgt, nicht übereinstimmen mit dem Refinement-Namen, in dem das LEAVE-Konstrukt textuell enthalten ist.

  Hierbei ist zu beachten, daß wie in dem Beispiel auf der vorigen Seite "durchsuche alle Spalten dieser Zeile" in dem Refinement "wert gefunden" angewendet wird (d.h. nach textueller Ersetzung enthalten ist).

- Der Terminator kann eine Resultats-Angabe enthalten. Diese besteht aus einer Einheit, die einen Wert liefern muß.

Dabei ist wichtig, daß der Terminator selbst keinen Wert liefert. Vielmehr "veranlaßt" er, daß das zu beendende Refinement (bzw. Prozedur oder Operator) einen Wert liefert. Der Datentyp und das Accessrecht dieses Wertes ergibt sich aus dem Datentyp und dem Accessrecht der Einheit, die auf das Schlüsselwort WITH folgt.
Das hat zur Folge, daß Sprachelemente praktisch mit Werten verlassen werden können, obschon sie selbst keine Werte liefern können. So haben wir im letzten Beispiel eine geschachtelte Schleife mit einem Wert verlassen. Dies ist allerdings nur dann möglich, wenn diese Sprachelemente Teile eines Refinements (bzw. einer Prozedur oder eines Operators) sind.
Eine Balancierung des Accessrechts erfolgt, wenn ein Refinement ein Objekt liefert, das anderes Accessrecht hat als das Objekt, das mit dem Terminator angegeben wurde. In diesem Fall ist das Accessrecht immer CONST.

9. Die starke Datenabstraktion

Die Definition neuer Datentypen bietet die Möglichkeit, problemorientierte Datentypen in ELAN einzuführen und Operationen (vergl. Kap 10.) für diese zu definieren. Dabei werden sie auf die einfachen, in ELAN vorgegebenen Datentypen (vergl. Kap. 3. und 5.) oder andere, bereits neudefinierte Datentypen zurückgeführt. Die Kenntnis über die Komponenten der neudefinierten Datentypen, d.h. diejenigen Datentypen, auf die der neudefinierte Datentyp zurückgeführt wird, wird in ELAN Feinstruktur genannt.

Neudefinierte Datentypen werden auch abstrakte Datentypen genannt, wenn neben ihrer Definition auch Operationen (Prozeduren und Operatoren) auf diesen Typ realisiert werden. Außerdem soll der direkte Zugriff auf die Feinstruktur eines Objekts dieses Datentyps nicht möglich sein und die algorithmische Realisierung der Operationen dem Benutzer verborgen bleiben. Die Anforderungen lassen sich in ELAN aber nur mittels des Modul-Konzepts realisieren, das wir im nächsten Kapitel erklären.

Im Gegensatz zu der schwachen Datenabstraktion wird durch die starke Datenabstraktion nicht nur ein neuer Name, sondern auch ein neuer Datentyp eingeführt, der im Allgemeinen andere Eigenschaften als die Realisierung hat. Abstrakte Datentypen werden benutzt, um Algorithmen möglichst unabhängig von den verwendeten Datenstrukturen zu machen.

9.1. Die Definition neuer Datentypen

Im Gegensatz zu den Abkürzungsvereinbarungen für Datentypen (LET-Konstrukt, vergl. Kap. 7.2.) führt eine Definition eines neuen Datentyps einen neuen Typ in die Programmiersprache ELAN ein, auf dem keine Operationen definiert sind.

Beispiel:

TYPE GEHALT = REAL

Mit der Definition des neuen Datentyps GEHALT wird gleichzeitig die Feinstruktur, d.h. die Realisierung dieses neuen Datentyps festgelegt (hier: REAL). Soll nun ein Datenobjekt des Typs GEHALT bereitgestellt werden, so muß wie bei den einfachen, vorgegebenen Datentypen dieses Objekt deklariert und mit einem Namen versehen werden:

GEHALT VAR mein gehalt

Man beachte also den Unterschied zwischen der Definition von Datentypen und der Deklaration von Datenobjekten als Objekt dieses Typs. Die neudefinierten Datentypen können genauso wie die vorgegebenen Datentypen verwendet werden, z.B. in der Deklaration von Reihungen oder Strukturen.

Beispiel:

ROW 100 GEHALT VAR gehaltsliste

Neue Datentypen werden folgendermaßen definiert:

- Dem Schlüsselwort TYPE folgt der Bezeichner des neudefinierten Datentyps, der ebenfalls als Schlüsselwort dargestellt wird.

- Hinter dem Gleichheitszeichen wird ein Datentyp angegeben.

Dabei kann man drei Arten der Feinstruktur unterscheiden, die wir in dem folgenden Abschnitt genauer besprechen werden.

Um mit neudefinierten Datentypen arbeiten zu können, brauchen wir folgende Operationen:

- Eine Möglichkeit, um Denoter für neudefinierte Datentypen zu schreiben. Dies ist notwendig, weil alle bis jetzt besprochenen Denoter nicht vom gleichen Typ wie der neudefinierte Datentyp sind.
- Eine Möglichkeit, auf die Feinstruktur neudefinierter Datentypen zuzugreifen. Dies ist notwendig, da keine Operationen auf den neudefinierten Datentypen vorhanden sind, aber (hoffentlich) auf deren Komponenten.
- Eine Möglichkeit, Reihungen und Strukturen von neudefinierten Datentypen (wie oben erwähnt) auch zu subskribieren bzw. zu selektieren.

Nötig ist weiter, daß auch Operationen auf den neudefinierten Datentypen als Ganzes vereinbart werden können.

9.1.1. Denoter für abstrakte Datentypen (Konstruktor)

Denoter für abstrakte Datentypen kann man in ELAN formulieren, indem man den Konstruktor (vergl. Abschnitt 5.2.2.) verwendet.

Beispiel:

```
TYPE GEHALT = REAL;
GEHALT CONST mein gehalt :: GEHALT : (10000.0)
```

Der Konstruktor (hier: GEHALT : (10000.0)) für neudefinierte Datentypen wird analog den Konstruktoren für Strukturen gebildet. Zur Kennzeichnung wird der neudefinierte Datentyp benutzt. Konstruktoren können auch geschachtelt auftreten:

```
(* Typ-Definition: *)
TYPE ALTER = INT;
TYPE PERSON = STRUCT (TEXT name, ALTER age);

(* Datenobjekt-Deklaration: *)
PERSON VAR rainer :: PERSON: ("Hahn", ALTER: (45))
```

9.1.2. Zugriff auf die Feinstruktur (Konkretisierung)

Der Konkretisierer ("concretizer") gestattet den Zugriff auf die Feinstruktur von neudefinierten Datentypen.

```
TYPE GEHALT = REAL;

REAL VAR netto;
GEHALT VAR brutto :: GEHALT : (10000.0);

netto := CONCR (brutto)/100.0 * 10.0
(* d.h. 10 Prozent *)
```

Der Konkretisierer erlaubt also eine typenmäßige Umbetrachtung eines Datenobjektes, ohne daß sein Wert geändert wird. D.h. man steigt eine Ebene "hinunter" und arbeitet dann auf der Feinstruktur des Datentyps.

Eine Angabe eines Konkretisierers besteht aus dem Schlüsselwort CONCR, gefolgt von einer geklammerten Einheit, die einen abstrakten Datentyp liefern muß. Der Datentyp des Konkretisierers ist der Datentyp der Feinstruktur. Das Accessrecht wird durch die Anwendung eines Konkretisierers nicht verändert. Daher kann ein Datenobjekt mit dem Accessrecht VAR nach Anwendung des Konkretisierers auf der linken Seite einer Zuweisung erscheinen:

CONCR (brutto) := 999.9

Mit dem Konkretisierer sind ebenso wie bei dem Konstruktor geschachtelte Konstruktionen möglich.

Der Konkretisierer ist notwendig für neudefinierte Datentypen, deren Feinstruktur (unter Umständen nach mehrmaliger Anwendung des Konkretisierers) von einfachem Datentyp ist. Besteht die Feinstruktur hingegen aus einer Reihung oder Struktur, so kann eine implizite Konkretisierung durch Subskription oder Selektion (vergl. Abschnitt 9.2.) erfolgen.

Der Zugriff auf die Feinstruktur ist nur in dem Paket (vergl. Abschnitt 11.2.2.) erlaubt, in dem sie definiert wurde.

9.2. Beispiele für Feinstrukturen neudefinierter Datentypen

In diesem Abschnitt wollen wir die verschiedenen Möglichkeiten der Feinstruktur der neudefinierten Datentypen sowie die jeweiligen Formen der Konkretisierer und Konstruktoren aufzeigen.

9.2.1. Feinstruktur: Einfache Datentypen

Definition

a) *TYPE ALTER = INT*

Deklarationen

b1) *ALTER VAR rainer*
b2) *ROW 2 ALTER VAR ehepaar*
b3) *STRUCT (TEXT name, ALTER age) VAR person*

Zugriff auf die Feinstrukturen

c1) *CONCR (rainer)*
c2) *CONCR (ehepaar [1])*
c3) *CONCR (person.age)*

Denoter

d1) *ALTER: (35)*
d2) *[ALTER: (34), ALTER: (35)]*
d3) *STRUCT (TEXT name, ALTER age): ("Hahn", ALTER: (35))*

9.2.2. Feinstruktur: Reihung

Definition

a) *TYPE TABELLE = ROW 5 INT*

Deklarationen

b1) *TABELLE VAR feld 1*
b2) *ROW 2 TABELLE VAR feld 2*
b3) *STRUCT (INT index, TABELLE liste) VAR element*

Zugriff auf die Feinstrukturen

c1) *feld 1 [5]*
c2) *feld 2 [1] [5]*
c3) *element. liste [5]*

Denoter

d1) *TABELLE : (1,2,3,4,5)*
d2) *[TABELLE: (1,2,3,4,5), TABELLE: (1,2,3,4,5)]*
d3) *STRUCT (INT index, TABELLE liste): (3, TABELLE: (1,2,3,4,5))*

9.2.3. Feinstruktur: Struktur

Definition

a) *TYPE PERSON = STRUCT (TEXT name, INT alter)*

Deklarationen

b1) *PERSON VAR erster mann*
b2) *ROW 2 PERSON ehepaar*
b3) *STRUCT (PERSON frau, INT anzahl aepfel) VAR eva*

Zugriff auf die Feinstrukturen

c1) *erster mann.name (* bzw. *) erster mann. alter*
c2) *ehepaar [1]. name (* bzw. *) ehepaar [1]. alter*
c3) *eva.frau. name (* bzw. *) eva.frau. alter (* bzw. *) eva.anzahl aepfel*

Denoter

d1) *PERSON : ("adam", 17)*
d2) *[PERSON : ("adam", 17), PERSON: ("eva", 32)]*
d3) *STRUCT (PERSON frau, INT anzahl aepfel): (PERSON: ("eva", 34), 100)*

10. Die starke algorithmische Abstraktion

Als starke algorithmische Abstraktion werden in ELAN Prozeduren und neudefinierte Operatoren bezeichnet. Prozeduren können mit oder ohne Parameterteil und mit oder ohne Resultat geschrieben werden, während Operatoren mit einem bzw. zwei Parametern (für monadische bzw. dyadische Operatoren) vereinbart werden.

Ineinander geschriebene (geschachtelte) Prozeduren bzw. Operatoren sind nicht möglich. Einige Prozeduren und Operatoren werden standardmäßig vorgegeben, insbesondere für die Ein-/Ausgabe und Operationen auf einfachen Datentypen.

Man unterscheidet zwischen der Deklaration und dem Aufruf einer Prozedur oder eines Operators. Dabei besteht die Deklaration aus dem Prozedur- bzw. Operatortext, der eine Formulierung des entsprechenden Algorithmus in ELAN ist, und der Aufruf aus dem Bezeichner der Prozedur bzw. des Operators mit den ggfs. dazugehörigen aktuellen Parametern oder Operanden.

10.1. Prozeduren

Prozeduren werden - im Gegensatz zu Refinements (vergl. Kap. 8) - verwendet, wenn

- ein Programmteil benannt und als "black box" verwendet werden soll,
- ein bis auf eventuelle Parameter identischer Programmteil in verschiedenen Stellen im Programmtext vorkommt,
- lokale Variablen verwendet werden sollen, d.h. Datenobjekte, die nur für die Dauer der Prozedurelaboration vorhanden sein sollen,
- Programmteile rekursiv sein sollen.

Die wichtigste Eigenschaft von Prozeduren - hinsichtlich der Abstraktion - ist jedoch, daß ein Programmierer nur über den Effekt einer Prozedur ("was" macht die Prozedur) Bescheid wissen muß und nicht, wie die Prozedur programmiert ist ("wie" ist der Algorithmus). Prozeduren können ebenso wie neudefinierte Operatoren (vergl. 10.2.) daher als "black box" angesehen werden.

10.1.1. Prozeduren ohne Parameter

Prozeduren ohne Parameter und ohne Resultat sind die einfachsten Prozeduren in ELAN:

Beispiel:

```
(* Globale Variablen: *)
INT VAR a :: 1, b :: 2;

(* Prozedur-Deklaration: *)
PROC vertausche:
   INT CONST hilf :: a;  (* lokales Datenobjekt *)
   a := b;
   b := hilf
END PROC vertausche;
:
:
(* Prozeduraufruf: *)
vertausche;
:
:
(* nochmals: *)
vertausche;
:
:
```

Eine Prozedur (ohne Parameter, ohne Resultat) deklariert man folgendermaßen:

- Dem Schlüsselwort PROC folgt der Name der Prozedur, der mit einem Doppelpunkt ("colon") abgeschlossen wird.
- Dem Doppelpunkt folgt der Prozedurrumpf ("proc body"). Der Prozedurrumpf kann aus Deklarationen (aber nicht: Typ-Definitionen, die außerhalb von Prozeduren geschrieben werden müssen und anderen Prozeduren), Einheiten (Units) und Refinements bestehen.
- Der Prozedurrumpf wird durch die Schlüsselworte END PROC, die von dem Namen der Prozedur gefolgt werden, abgeschlossen.

An dem vorherigen Beispiel kann auch der Unterschied von globalen (auch Paket-Variablen) und lokalen Variablen demonstriert werden.

Die globalen Variablen, die auch Paket-Variablen genannt werden, sind diejenigen Variablen, die außerhalb von Prozeduren deklariert werden und deshalb auch für die Prozedur Gültigkeit haben, d.h. auch innerhalb der Prozedur aus angesprochen werden können (hier: "a" und "b").

Lokale Daten werden nur innerhalb von Prozeduren deklariert und haben Gültigkeit in der Prozedur, in der sich ihre Deklaration befindet. Sie können ausschließlich innerhalb dieser Prozedur angesprochen werden und belegen erst dann Speicherplatz, wenn die Prozedur aufgerufen wird.

Der Aufruf einer Prozedur besteht aus dem Bezeichner der Prozedur. Diese Art des Prozeduraufrufs stellt eine Einheit dar.

10.1.2. Prozeduren mit Parametern

Soll ein Algorithmus unabhängig von bestimmten Variablen, Werten bzw. Prozeduren formuliert werden, so verwendet man Prozeduren mit Parametern. Parameter können sowohl Datenobjekte als auch Prozeduren sein.

10.1.2.1. Datenobjekte als Parameter

```
(* Globale Variablen: *)
INT VAR a :: 1, b :: 2, c :: 3;

(* Prozedur-Deklaration: *)
PROC vertausche (INT VAR x, y):
    INT CONST hilf :: x;  (* lokale Daten* )
    x := y;
    y := hilf
END PROC vertausche;

(* Prozeduraufrufe: *)
vertausche (a, b);
vertausche (b, c)
```

Eine Prozedur mit Parametern wird wie eine Prozedur ohne Parameter geschrieben (siehe vorigen Abschnitt), mit der Ausnahme, daß die Parameter in Klammern hinter dem Prozedurnamen (Parameter-Spezifikation) angegeben werden. Parameter werden hinsichtlich ihrer Anwendung unterschieden:

a) die formalen Parameter sind diejenigen Parameter, die in der Prozedurdeklaration (in Prozedurkopf und -rumpf) erscheinen. Sie dienen sozusagen als "Statthalter" für die einzusetzenden aktuellen Parameter.

b) Die aktuellen Parameter sind diejenigen Parameter, die bei dem Aufruf einer Prozedur in der Parameterliste einzusetzen sind.

Man kann sich die formalen Parameter vor jeder Ausführung der Prozedur durch die aktuellen Parameter ersetzt denken.

Formale Parameter werden durch Typ, Accessrecht und Namen spezifiziert. Mehrere Parameter werden durch Kommata getrennt. VAR oder CONST geben an, ob ein Parameter im Prozedurrumpf verändert werden darf oder nicht, d.h. ob er z.B. auf der linken Seite eines Zuweisungsoperators erscheinen darf.

Welche aktuellen Parameter verwendet werden dürfen, hängt von der Spezifikation der formalen Parameter ab:

- Ein formaler CONST-Parameter darf im Prozedurrumpf nicht verändert werden. Darum kann man als aktuellen Parameter ein mit VAR vereinbartes Datenobjekt, aber auch einen Ausdruck oder einen Denoter einsetzen.

  Beispiel:

  ```
  PROC beispiel 1 (INT CONST par 1, REAL CONST par 2):
  :
  :
  END PROC beispiel 1;
  (* Beispiel für einen Aufruf: *)
  beispiel 1 (a*b+c,d)
  ```

- Ein formaler VAR-Parameter darf im Prozedurrumpf verändert werden. Daher dürfen als aktuelle Parameter in diesem Fall nur mit VAR vereinbarte Datenobjekte, aber keine Ausdrücke, die ein CONST-Datenobjekt liefern (da eine Zuweisung an einen solchen Ausdruck verboten wäre) verwendet werden.

  Beispiel:

  PROC beispiel 2 (INT VAR par 1):
  :
  :
  par 1 := ...
  :
  :
  END PROC beispiel 2;

  (Beispiel für einen Aufruf: *)*
  beispiel 2 (a)

Der Datentyp der Parameter kann jeder in ELAN verfügbare Datentyp sein, d.h. einfache Datentypen, Datenverbunde (Reihungen und Strukturen) und neudefinierte Datentypen. Dabei ist es wichtig, daß die aktuellen und die formalen Parameter den gleichen Datentyp besitzen.

10.1.2.2. Prozeduren als Parameter

In einigen Anwendungsfällen ist es notwendig, Prozeduren als Parameter zu übergeben. Das ist in ELAN ebenfalls erlaubt. Eine Prozedur als formaler Parameter wird folgendermaßen spezifiziert:

- Angabe des Datentyps, den die Prozedur gegebenenfalls liefern soll,
- Angabe des Schlüsselwortes PROC,
- Angabe der geklammerten Parameterliste (ohne Bezeichner), falls die formale Prozedur Parameter besitzen soll,
- Angabe des Bezeichners der formalen Prozedur.

Beispiel:

PROC zeichne (REAL PROC (REAL CONST) funktion):

Dabei ist auf die unterschiedliche Notation zu achten, denn der Bezeichner der formalen Prozedur wird nicht nach dem Schlüsselwort PROC, sondern erst am Schluß angegeben.

Wir sehen also, daß die Angabe PROC (REAL CONST) im obigen Beispiel für das Accessrecht VAR oder CONST bei "einfachen" Parametern steht. Somit erhalten wir ein erweitertes Accessrecht bei Parametern für Prozedur-Parameter. Es bedeutet, daß eine Prozedur (im Prozedurrumpf) ausgeführt werden darf. Ein aktueller Prozedurparameter besteht nur aus dem Bezeichner der Prozedur, also ohne dessen Parameterliste.

Beispiel:

```
PROC kurve (REAL PROC (REAL CONST) trigonometrische funktion,
            REAL CONST anfang, ende, INT CONST anzahl):
    REAL VAR x :: anfang,
             y,
             schrittweite :: (ende - anfang)/real (anzahl);
    INT VAR i;
    FOR i FROM 1 UPTO anzahl REP
        y := trigonometrische funktion (x);
        markiere im koordinatensystem (x,y);
        x := anfang + i * schrittweite
    END REP
END PROC kurve

(* Beispiel für den Aufruf *)
kurve (sin, 0.0, 2.0 * pi, 50);
kurve (tan, -pi/4.0, pi/4.0, 100)
```

Bei dem Aufruf einer Prozedur mit einem Prozedur-Parameter ist zu beachten, daß die Deklaration der Prozedur, die übergeben wird, der Spezifikation der formalen Prozedur entspricht. Das bedeutet im obigen Beispiel, daß die Prozedur "sin" mit REAL PROC sin (REAL CONST ...) deklariert sein muß. Ferner ist wichtig, daß die aktuelle Parameterliste in Anzahl, Datentyp und Accessrecht mit der spezifizierten übereinstimmt und daß die formale Prozedur im Prozedurrumpf ihrer Spezifikation entsprechend verwendet wird.

10.1.3. Generizität von Prozeduren

Generizität bedeutet, daß unterschiedliche Prozeduren bzw. Operatoren denselben Bezeichner tragen können. Die Identifizierung erfolgt in diesem Fall durch die Anzahl, Reihenfolge und Datentyp der jeweiligen Parameter bzw. Operanden. Ein gutes Beispiel hierfür sind die Ein-/Ausgabe-Prozeduren ("get" und "put").

Bei der Ausgabe-Prozedur "put" will man alle vorgegebenen, einfachen Datentypen ausgeben. Für neudefinierte Datentypen müssen neue Prozeduren geschrieben werden. Von der Programmiersprache ELAN sind u.a. folgende Prozeduren vorgegeben (wir schreiben hier nur die "Prozedur-Köpfe"):

```
PROC put  (TEXT CONST wert):
PROC put  (INT  CONST wert):
PROC put  (REAL CONST wert):
PROC put  (BOOL CONST wert):
```

Für die Identifizierung generischer Prozeduren wird nur der Typ, aber nicht das Accessrecht der Parameter oder ein evtl. Resultat herangezogen.

10.1.4. Werte liefernde Prozeduren

Prozeduren dürfen in ELAN auch Werte liefern. Solche Prozeduren heißen in anderen Programmiersprachen auch Funktionen.

Beispiel:

```
REAL PROC minimum (REAL CONST a,b):
   IF a < b THEN a ELSE b END IF
END PROC minimum
```

Werte liefernde Prozeduren können in Ausdrücken aufgerufen und ebenfalls zur Initialisierung verwendet werden ("Denotierungs-Prozeduren").

Bei Prozeduren, die Werte liefern, ist folgendes zu beachten:

- Die letzte Einheit des Prozedurrumpfes muß einen Wert liefern.
- Der Typ des gelieferten Wertes muß mit dem Typ der Resultatsangabe übereinstimmen. Der Datentyp des Resultats wird vor dem Schlüsselwort PROC angegeben.
- Eine Prozedur darf jeden beliebigen Datentyp liefern, d.h. einfache, (vorgegebene) Typen, Datenverbunde (Reihungen und Strukturen) sowie neudefinierte Datentypen.

Beispiel:

```
TYPE KOMPLEX = STRUCT (REAL re, im);

KOMPLEX PROC summe (KOMPLEX CONST links, rechts):
   KOMPLEX: (links.re + rechts.re, links.im + rechts.im)
END PROC summe;

ROW 2 KOMPLEX PROC komplex vektor (KOMPLEX CONST a,b):
   [a,b]
END PROC komplex vektor
```

Denotierungs-Prozeduren sind Prozeduren, die einen (eventuell auch abstrakten) Wert liefern, der als Denoter Verwendung findet. Sie werden oft in Initialisierungen verwendet. Insbesondere sind Denotierungs-Prozeduren die einzige Möglichkeit, Objekte neudefinierter Datentypen zu initialisieren, sofern diese außerhalb des sie definierenden Pakets verwendet werden sollen. Die Deklaration dieser Prozeduren muß in dem Paket vorgenommen werden, in dem auch der entsprechende Datentyp definiert wird.

Beispiel:

```
TYPE VEKTOR = ROW 100 INT;

VEKTOR PROC init (INT CONST wert):
   VEKTOR VAR vek;
   INT VAR index;
   FOR index FROM 1 UPTO 100 REP
       vek [index] := wert
   END REP;
   vek
END PROC init;
:
VEKTOR CONST eins vektor :: init(1)
```

10.2. Operatoren

In ELAN hat man die Möglichkeit, für neudefinierte Datentypen auch Operatoren zu definieren. Die neudefinierten Datentypen und Operationen (Prozeduren und Operatoren) werden als abstrakte Datentypen bezeichnet und stellen in einem Modul zusammengefaßt, eine Spracherweiterung dar. Benutzt man ausschließlich diese Erweiterung, so stellt diese eine Fachsprache dar.

Normalerweise liefern Operatoren ein Resultat, da sie zumeist in Ausdrükken verwendet werden. Das Accessrecht dieser Werte ist immer CONST. Es gibt aber auch manchmal gewisse Anwendungen, in denen Operatoren kein Resultat liefern (wie z.B. ":=").

Die Namen von Operatoren (Schlüsselworte, ein oder zwei Sonderzeichen) können mit bereits vorhandenen Operator-Namen übereinstimmen. In diesem Fall muß jedoch die Anzahl, die Reihenfolge oder der Datentyp der Parameter unterschiedlich sein (generische Operatoren). Die aktuellen Parameter von Operatoren werden - in Anlehnung an die vorgegebenen Operatoren - Operanden genannt.

10.2.1. Neudefinierte monadische Operatoren

Monadische Operatoren haben nur einen Operanden, der bei der Anwendung rechts vom Operatorsymbol stehen muß. Als Beispiel für einen neudefinierten monadischen Operator verwenden wir hier einen Operator aus den Standardpaketen (vergl. Kap. 13):

```
INT OP ABS (INT CONST argument):
    IF argument  >= 0
    THEN argument
    ELSE -argument
    END IF
END OP ABS
```

Wie wir sehen, unterscheidet sich eine Operator-Deklaration nicht von einer Prozedur-Deklaration, mit der Ausnahme der Schlüsselworte OP bzw. END OP und dem Bezeichner des Operators. Die Regeln der Prozedur-Deklaration können also entsprechend angewendet werden. Das bedeutet, daß in einem Operatorrumpf lokale Datenobjekte deklariert und Refinements geschrieben werden können. Es ist die Möglichkeit gegeben, rekursive Operatoren zu formulieren.

Bei dem Operatornamen gibt es drei Möglichkeiten:

a) Ein Schlüsselwort ("bold", vergl. 2.1.).

b) Ein beliebiges Sonderzeichen, sofern diese Zeichen nicht als Delimiter verwendet werden (z.B. ",;,),(, Komma usw.).

Beispiel:

```
TYPE KOMPLEX = STRUCT (REAL re, im);
KOMPLEX VAR compl 1, compl 2;
:
:
KOMPLEX OP - (KOMPLEX CONST compl):
   KOMPLEX: (-compl.re, -compl.im)
END OP -;
:
:
compl 1 := - compl 2;
:
:
```

c) Eine Kombination von zwei Sonderzeichen. Diese kann nur dann verwendet werden, wenn sie bereits in ELAN existiert. Somit dürfen folgende Kombinationen verwendet werden:

:=, <>, <=, >=, **

Andere Kombinationen sind unzulässig.

10.2.2. Neudefinierte dyadische Operatoren

Dyadische Operatoren haben zwei Operanden, die bei der Anwendung links und rechts von dem Operatorsymbol stehen müssen.

Als Beispiel für einen neudefinierten dyadischen Operator zeigen wir wiederum einen Operator aus den Standardpaketen (vergl. Kap. 13.).

```
OP INCR (INT VAR dazu, INT CONST wert):
   dazu := dazu + wert
END OP INCR;
:
:
INT VAR index :: 1;
WHILE index < max elemente REP
   :
   :
   index INCR 1
END REP
```

Für die Bildung von Namen für neudefinierte dyadische Operatoren gelten die gleichen Regeln wie für neudefinierte monadische Operatoren (vergl. vorigen Abschnitt).

10.2.3. Die Priorität neudefinierter Operatoren

Jeder Operator (also auch ein neudefinierter) hat in ELAN genau eine, nicht änderbare Priorität. Monadische Operatoren haben immer die höchste Priorität (Priorität 9), d.h. monadische Operatoren werden in einem Ausdruck vor dyadischen Operatoren ausgeführt.

Bei dyadischen Operatoren haben wir zwei Fälle zu unterscheiden:

- das oder die aufeinanderfolgenden Zeichen (wenn es sich bei dem Operator um zwei Sonderzeichen oder ein Schlüsselwort handelt) stimmen mit einem der Operatorzeichen überein, die in der Tabelle im Abschnitt 4.3. angegeben wurden. In diesem Fall erhält der Operator die für diese(s) Zeichen zugeordnete Priorität. Ein neudefinierter "*"-Operator (Priorität 7) wird also in einem Ausdruck vor einem neudefinierten dyadischen "+"-Operator (Priorität 6) ausgeführt. Für das folgende Beispiel nehmen nehmen wir an, daß der "+"- und der "*"-Operator für TEXT bzw. INT und TEXT neudefiniert werden und nicht in den Standardpaketen (vergl. Kap. 13.) vorhanden wären.

  Beispiel:

  ```
  TEXT CONST plus :: "+", minus :: "-";
  :
  put (plus + 4 * minus + plus);
  :
  (* entspricht *)
  put ("+---+")
  ```

- Alle anderen dyadischen neudefinierten Operatoren erhalten die Priorität 2.

  Beispiel:

  ```
  TEXT VAR t :: "abc";
  :
  t CAT (t SUB LENGTH t)
  ```

  Hier darf die Klammer nicht fortgelassen werden, da CAT und SUB die gleiche Priorität besitzen und die Operatoren deshalb von links nach rechts abgearbeitet würden. Da CAT jedoch keinen Wert liefert, hingegen SUB als linken Operanden ein TEXT-Datenobjekt erwartet, würde das Fortlassen der Klammer zu einem syntaktischen Fehler führen. Der obige Ausdruck hängt übrigens das letzte Zeichen von t an t an.

Der Grund für diese Vorschrift liegt auf der Hand, denn für die üblichen Operatoren wird die gewohnte Priorität (auch für andere Operanden-Datentypen) beibehalten. Bei anderen neudefinierten Operatoren wird immer die gleiche Priorität verwendet, die niedriger - mit Ausnahme der Zuweisung - als alle übrigen Operatoren ist. Diese Vorschrift über die Priorität von Operatoren hat sich - obwohl sie auf den ersten Blick kompliziert erscheint - als sehr praktisch erwiesen.

Der Hauptnachteil dieser Lösung ist, daß ungeübte Benutzer oft nicht unterscheiden können, ob ein Operator vorgegeben oder durch Standardpakete realisiert (und damit neudefiniert) ist.

Beispiel:

```
TEXT CONST zeile :: "das ist eine zeile";
:
:
IF zeile SUB 2 = "a" THEN ... ENDIF
(* gemeint ist:
IF (zeile SUB 2) = "a" THEN ... ENDIF *)
:
:
```

In diesem Fall erhält der Benutzer von dem ELAN-Compiler eine Fehlermeldung, die besagt, daß kein dyadischer Operator zwischen Operanden vom Datentyp INT und TEXT definiert ist. Gemeint ist der "="-Operator, der eine höhere Priorität als der neudefinierte SUB-Operator hat. Folgende Anweisung ist ebenfalls falsch:

```
zeile SUB 2 := "x"
```

Der Zuweisungsoperator hat zwar eine niedrigere Priorität als SUB, jedoch liefert "zeile SUB 2" durch den SUB-Operator ein CONST-Datenobjekt. Man muß also in diesem Fall mit der "replace"-Prozedur arbieten (vergl. Kap. 13).

10.3. Terminatoren für Prozeduren und Operatoren

Terminatoren ("LEAVE-Konstrukt") haben wir schon für Refinements kennengelernt. Terminatoren können jedoch auch für Operatoren bzw. Prozeduren verwendet werden, um diese vorzeitig zu beenden. Im Gegensatz zu der Verwendung von Terminatoren in Refinements muß ein Terminator zum Verlassen einer Prozedur oder eines Operators in der betreffenden Prozedur oder dem Operator geschrieben werden.

Beispiel:

```
INT OP ** (INT CONST basis, exponent):
   :
   :
   IF basis <> 0 AND exponent = 0
   THEN LEAVE ** WITH 1
   END IF;
   :
   :
END OP **
```

Terminatoren dürfen an jeder Stelle in einer Prozedur oder einem Operator geschrieben werden, an der eine Anweisung, d.h. eine Einheit, die keinen Wert liefert, stehen kann. Sie liefern jedoch selber keinen Wert, sondern verlassen lediglich das benannte Konstrukt (Prozedur, Refinement oder Operator) mit einem Wert. Operatoren und Prozeduren, die keinen Wert liefern, können nur Terminatoren ohne Resultatsangabe enthalten.

11. Pakete

Die Programmiersprache ELAN bietet die Möglichkeit, Programmteile (Module, in ELAN "PACKET" genannt) voneinander abzugrenzen und unabhängig von dem Rest eines Programms übersetzen zu lassen.

Mittels der Schnittstelle ("interface") des Pakets werden diejenigen Konstanten, Datentypen, Operatoren und Prozeduren bezeichnet, die dem nachfolgenden Programm, das wiederum aus Paketen bestehen kann, zur Verfügung gestellt werden sollen. Man kann also Datenobjekte und Operationen eines Pakets vor dem (unberechtigten bzw. fehlerhaften) Zugriff schützen, indem diese Objekte nachfolgenden Programmteilen nicht über die Schnittstelle "hinausgereicht" werden. Durch die Bereitstellung von Paketen können Fachsprachen, die auf einen speziellen Problemkreis zugeschnitten sind, geschaffen werden (vergl. 11.3.).

11.1. Module

Wir wollen am oftgezeigten*) (und deshalb berüchtigten) Beispiel eines Stacks die Realisierung eines Pakets in ELAN verdeutlichen:

```
PACKET stack handling DEFINES push, pop:

LET max = 1000;
ROW max INT VAR stack;
INT VAR stackpointer :: 1;
(*stackpointer zeigt immer auf den ersten freien Platz des stacks*)

PROC push (INT CONST wert):
   IF stackpointer > max THEN
      errorstop ("stack overflow")
      (*errorstop siehe Standardpakete*)
   ELSE
      stack [stackpointer] := wert;
      stackpointer INCR 1
   END IF
END PROC push;

PROC pop (INT VAR wert):
   stackpointer DECR 1;
   IF stackpointer < 1 THEN
      errorstop ("stack underflow")
   ELSE
      wert := stack [stackpointer]
   END IF
END PROC pop

END PACKET stack handling
```

---

*) "A lecture without stacks is like a day without sunshine".
J. Guttag

Wie wir in diesem Beispiel sehen, verbirgt man die tatsächliche Realisierung und den Zugriff auf den Stack vor dem nachfolgenden Programm ("hiding of information"). Den Schutzeffekt erreicht man durch die Angabe von PACKET bzw. END PACKET, wodurch auf alle Objekte, die sich innerhalb dieser Klammerung befinden, von außen nicht zugegriffen werden kann. Ausgenommen davon sind die Objekte, die man in der Schnittstelle des Pakets aufführt (hinter DEFINES). Dadurch erreicht man folgende wichtige Ziele:

a) Die Realisierung von Datenobjekten kann geändert werden, ohne daß nachfolgende Programme modifiziert werden müssen. Findet man etwa für eine spezielle Anwendung eine effizientere Realisierung eines Stacks (z.B. durch eine gekettete Liste), so ändert man diese und die dazugehörigen Operationen, aber nicht die Namen der Datenobjekte und Operationen. Dadurch ist es nicht erforderlich, daß ein Benutzer (d.h. derjenige, der einen Stack in seinem Programm verwendet) auch sein Programm abändern muß, da die Wirkungsweise der Operationen "push" und "pop" für ihn gleich bleiben (hoffentlich!).

b) Pakete können unabhängig voneinander geschrieben, übersetzt und ausgetestet werden. Das Prinzip der Modularität, das in ELAN durch die Pakete realisiert wird, erlaubt, daß verschiedene Programmierer an einem Programmier-Projekt zusammenarbeiten können.

c) Pakete erlauben es, Datenobjekte vor einem fehlerhaften Zugriff zu schützen. Z.B. ist die Anweisung

   stack [*3*] := ...

   außerhalb des Pakets "stack handling" nicht erlaubt.

d) Durch Pakete kann man den Vorrat an Operationen und Datentypen, die die Programmiersprache ELAN bietet, erweitern. Stellt man etwa das oben erwähnte Paket "stack handling" allen ELAN-Benutzern in vorübersetzter Form zur Verfügung, so hat man die Programmiersprache ELAN um die Operationen (Prozeduren) "push" und "pop" erweitert.

11.1.1. Formulierung eines Pakets

Ein Paket in ELAN wird durch das Schlüsselwort PACKET eingeleitet, dem der Bezeichner des Pakets folgen muß. Danach schließt sich die Schnittstelle (Interface) an. Das Interface besteht aus dem Schlüsselwort DEFINES, den Bezeichnern der Objekte (bei mehreren: durch Komma getrennt), die über die Schnittstelle "hinausgereicht" werden sollen, und einem abschließenden Doppelpunkt. Der anschließende Paketrumpf ("packet body") wird von dem Schlüsselwort END PACKET und dem Bezeichner des Pakets abgeschlossen. Mehrere Pakete werden voneinander jeweils durch ein Semikolon getrennt.

11.1.2. Schnittstelle eines Pakets

In dem Interface ("DEFINES-LISTE") werden die Bezeichner derjenigen Objekte aufgeführt, die dem nachfolgenden Programm zur Verfügung gestellt werden sollen. Folgende Bezeichner von Objekten dürfen in dem Interface aufgeführt werden:

- Die Bezeichner von CONST-Datenobjekten:

  PACKET trigometric DEFINES pi, ...:
  REAL CONST pi :: 3.14159;
  :
  :

  Der Datentyp des CONST-Datenobjekts wird dabei nicht automatisch mit über die Schnittstelle herausgegeben. Das ist wichtig für CONST-Datenobjekte neudefinierter Datentypen (der Datentyp muß mit angegeben werden).

- Die Bezeichner von Prozeduren und Operatoren:

  PACKET stackhandling DEFINES PUSH, pop:
  :
  :
  OP PUSH (INT CONST wert):
  :
  :
  END OP PUSH;
  PROC pop (INT VAR wert):
  :
  :
  END PROC pop
  :
  :

  Parameter und evtl. Resultatsangaben werden nicht im Interface angegeben. Es ist nicht erlaubt, Bezeichner gleichnamiger (generischer) Prozeduren bzw. Operatoren mehrfach aufzuführen.

- Die Bezeichner von neudefinierten Datentypen.

  Beispiel:

  PACKET complex numbers DEFINES KOMPLEX, ...:
  TYPE KOMPLEX = STRUCT (REAL re, im);

  :
  :

Bei der Angabe der Bezeichner in dem Interface spielt die Reihenfolge keine Rolle.

In dem Interface können folgende Objekte nicht aufgeführt werden:

- VAR-Datenobjekte,
- Namen von mit LET abgekürzte Datentypen und Denoter.

11.1.3. Paketrumpf ("packet body")

In dem Paketrumpf können in beliebiger Reihenfolge Deklarationen und Einheiten stehen.

Die Deklarationen können aus

- der Neudefinition von Datentypen (es ist möglich, auch mehrere verschiedene Datentypen in einem Paket zu definieren),
- Prozedur- und/oder Operatordeklarationen,
- Datenobjekt-Deklarationen und
- LET-Definitionen bestehen.

Die Einheiten ("units") können die in den vorherigen Kapiteln besprochenen Konstrukte (Abfragen, Auswahl, Wiederholung) enthalten. Verboten sind, außerhalb von Prozedur- und Operatordeklarationen, die Definition und Anwendung von Refinements sowie die Anwendung des Terminators, da der Terminator in keiner der erwähnten Einheiten verwendet werden kann.

Ein besonderes Paket, nämlich das letzte (das auch alleine erscheinen darf), wird "main packet" genannt. Es wird ohne PACKET, Interface und END PACKET geschrieben. Es legt den Anfang der Elaboration eines ELAN-Programms mit der ersten ausführbaren Einheit fest. Dieses "main packet" (vergl. 11.2.1) kann entweder aus den obenstehenden Deklarationen und/oder Anweisungen bestehen, oder aus einem Prozedurrumpf, der dann auch Refinements (einschließlich Terminator) enthalten darf. Jedoch dürfen Operator-, Prozedurdeklarationen sowie Typdefinitionen und Refinements außerhalb von Prozeduren im "main packet" nicht zusammen erscheinen.

11.2. Programmstruktur

In diesem Abschnitt wollen wir die Struktur von ELAN-Programmen darlegen. Dabei gehen wir auf die Reihenfolge von Paketen, auf die möglichen Gültigkeitsbereiche und auf Überdeckungen von Datenobjekten ein.

11.2.1. Reihenfolge und Struktur von Paketen

Ein ELAN-Programm besteht entweeder aus null oder mehr Paketen, die von einem "main packet" gefolgt werden (für ein Beispiel vergl. 11.3.). Folgenden Programmstrukturen sind daher möglich:

a) Prozedurrumpf als "main packet".
   Hier dürfen nur lokale Variablen (des Prozedurrumpfs) deklariert und nur LET-Definitionen benutzt werden. Weiterhin können Refinements eingesetzt werden. Verboten sind jedoch Prozedur-, Operator-Deklarationen und die Definition neuer Datentypen.

Beispiel:

```
REAL VAR x, ...;
eingabe;
berechne;
ausgabe.

eingabe:
:
berechne:
:
ausgabe:
:
```

b) Ein Paketrumpf als "main packet".
In diesem Fall ist die Deklaration von Prozeduren und/oder Operatoren möglich, sowie die Definition neuer Datentypen. Verboten sind Refinements ausserhalb von Prozeduren. Die Ausführung eines solchen "main packet" erfolgt durch die Abarbeitung der ersten ausführbaren Anweisung. Dazu gehören auch Initialisierungen.

Beispiel:

```
WHILE NOT eof REP
   eingabe
END REP

PROC eingabe:
   REAL VAR x, ...;
   :
   berechne (x);
   :
END PROC eingabe;

PROC berechne (REAL CONST y):
   REAL VAR z, ...;
   :
   ausgabe (z);
   :
END PROC berechne;

PROC ausgabe (REAL CONST wert):
   :
END PROC ausgabe
```

c) Mehrere Pakete, gefolgt von einem "main packet".

Beispiel:

PACKET eins DEFINES ...:

:

END PACKET eins;

:

PACKET n paket DEFINES ...:

:

END PACKET n paket;
(jetzt das "main packet" *)*

:

Bei dieser Struktur ist folgendes zu beachten:

Die Ausführung dieses ELAN-Programms beginnt mit der Abarbeitung des "main packet". Die Abarbeitung des "main packet" erfolgt in der Reihenfolge der ausführbaren Anweisungen, einschließlich der Initialisierungen bei Deklarationen. Wird dabei ein Objekt aus einem vorhergehenden Paket verwendet, werden die Objekte dieses Pakets initialisiert und entsprechend ausgeführt. D.h. die Reihenfolge der Ausführung von Paketen ist nicht festgelegt, sondern hängt von dem jeweiligen Programm ab. Streng genommen ist jedes ELAN-Programm vom Typ c), weil jedem Programm die Standardpakete implizit voranstehen.

11.2.2. Gültigkeitsbereich von Objekten

Der Gültigkeitsbereich von Objekten, der in ELAN "range" genannt wird, gibt an, an welchen Programmstellen Objekte angesprochen werden können. D.h. der Gültigkeitsbereich legt fest, welche Bezeichner aus einer Deklaration für eine bestimmte Programmstelle Gültigkeit haben. ELAN unterscheidet drei Gültigkeitsbereiche, wobei immer der kleinste umfassende Gültigkeitsbereich für ein Objekt gilt:

a) Prozedurrumpf ("proc body") (bzw. Operatorrumpf), ist der kleinste mögliche Gültigkeitsbereich. Von einem Prozedurrumpf können angesprochen werden

- Datenobjekte und LET-Vereinbarungen des Pakets, in dem die Prozedur vereinbart wird,
- Operatoren bzw. Prozeduren desjenigen Pakets, in dem der Prozedurrumpf sich befindet.
- Die Feinstruktur neudefinierter Datentypen, die in dem Paket definiert wurden, in dem sich die Prozedur befindet
- lokale Datenobjekte, Refinements und LET-Bereinbarungen der Prozedur,
- formale Parameter der Prozedur, die den Prozedurrumpf einschließt,

- Objekt aus vorangegangenen Paketen, die durch die entsprechenden Interfaces "herausgereicht" werden.

Nicht angesprochen werden können

- Objekte, die sich in parallelen Prozeduren befinden; dazu gehören lokale Datenobjekte, Refinements und formale Parameter,
- Objekte vorangehender Pakete, die nicht im jeweiligen Interface aufgeführt werden,
- die Feinstruktur neudefinierter Datentypen, die in anderen Paketen definiert werden,
- Objekte nachfolgender Pakete.

b) Paketrumpf ("packet body"). Von einem Paketrumpf können angesprochen werden

- diejenigen Datenobjekte, die sich in diesem Paket befinden und nicht lokale Daten bzw. formale Parameter einer Prozedur sind.
- LET-Vereinbarungen des Pakets,
- Prozeduren und Operatoren des betreffenden Pakets,
- die Feinstruktur neudefinierter Datentypen, die in diesem Paket definiert werden,
- Objekte vorangegangener Pakete, die in diesen Paketen in dem Interface aufgeführt wurden.

Nicht angesprochen werden können

- Objekte, die sich in Prozeduren befinden (lokale Daten, LET-Vereinbarungen, formale Parameter und Refinements),
- Objekte von vorangegangenen Paketen, die dort nicht in den entsprechenden Interfaces angeführt wurden,
- Objekte nachfolgender Pakete.

c) Programm. Für Objekte, die in der Schnittstelle eines Pakets aufgeführt sind, wird der Gültigkeitsbereich über den Paketrumpf hinaus auf nachfolgende Pakete erweitert; er heißt in ELAN "extended range" Es kommt also auf die textuelle Reihenfolge der Pakete an, da ein Objekt aus dem "extended range" im Interface eines vorangegangenen Pakets aufgeführt sein muß.

Die geschilderten Gültigkeitsbereiche beschränken nicht nur die Programmstellen, an denen bestimmte Bezeichner verwendet werden dürfen, sondern verhindern auch, daß gleichnamige Bezeichner in einem Gültigkeitsbereich deklariert bzw. spezifiziert werden dürfen mit der Ausnahme generischer Prozeduren und Operatoren. Selbstverständlich sind gleiche Bezeichner in parallelen Gültigkeitsbereichen (z.B. parallelen Prozeduren) erlaubt. Den Fall, daß ein Bezeichner eines umfassenden Gültigkeitsbereiches (z.B. aus dem "extended range") mit dem eines untergeordneten (z.B. Prozedurrumpf) in Konflikt gerät, behandeln wir im folgenden Abschnitt.

11.2.3. Überdeckungen

Überdeckungen treten dann auf, wenn Bezeichner von Objekten umfassender Gültigkeitsbereiche mit Bezeichnern eingeschlossener Gültigkeitsbereiche namensgleich sind. In anderen Programmiersprachen gibt es mehrere Möglichkeiten dieser Überdeckungen. In diesem Fall tritt eine Einschränkung des Gültigkeitsbereiches einer Vereinbarung ein, denn der Bezeichner des eingeschlossenen Gültigkeitsbereiches setzt den des umfassenden außer Kraft (z.B. Blockschachtelung in ALGOL 60, Schachtelung von Prozeduren in PASCAL). Dies kann unter Umständen ein gewünschter Effekt sein, führt jedoch leicht zu Fehlern. Deshalb sind aus Sicherheitsgründen in ELAN nur Namensüberdeckungen aus dem "extended range" erlaubt. D.h. eine Namensüberdeckung tritt auf, wenn für ein Objekt aus einem vorangegangenen Paket, das über ein Interface herausgereicht wurde ("extended range"), der gleiche Name verwendet wird wie für ein Objekt in einem nachfolgenden Paket.

In diesem Fall gilt für dieses Paket das zuletzt deklarierte Objekt. D.h. das Objekt aus dem "extended range" wird überdeckt und ist nicht gültig für dieses Paket.

Beispiel:

```
PACKET ausgaberoutinen DEFINES put:
PROC put (REAL CONST wert):
(* Standard-Ausgabe in Gleitpunktdarstellung *)
:
END PROC put;
:
END PACKET ausgaberoutinen;
:
PROC put (REAL CONST zahl):
(* Ausgabe in Festpunktdarstellung *)
:
END PROC put;
:
```

Hier "überdeckt" die zuletzt geschriebene Prozedur put die gleichnamige Prozedur aus den "extended range". Solch ein Vorgehen ist u. U. sinnvoll, wenn man eine eigene "sin"-Prozedur schreiben will, die (vielleicht) effizienter arbeitet als die in den Standardpaketen vorhandene Prozedur.

Andere Namensüberdeckungen sind in ELAN nicht möglich. Insbesondere dürfen in einem Gültigkeitsbereich Datenobjekte und Prozeduren sowie globale und lokale Datenobjekte (wie etwa in anderen Programmiersprachen) nicht den gleichen Namen haben. Gleiche Namen können jedoch in folgenden Fällen verwendet werden:

- Gleiche Prozedur- oder Operatornamen, die sich in den Typen und/oder Anzahl sowie Reihenfolge der formalen Parameter unterscheiden (generische Prozeduren oder Operatoren).
- Gleiche Namen in parallelen Gültigkeitsbereichen. Z.B. darf in einer Prozedur der Refinement-Name "x" verwendet werden, der in einer parallelen Prozedur als eine INT-Variable vereinbart ist.

Diese beiden Fälle sind jedoch keine Namensüberdeckungen und bewirken keine Einschränkung des Gültigkeitsbereiches einer Vereinbarung.

11.3. Abstraktionsebenen

Wie wir schon in vorhergehenden Abschnitten dieses Kapitels erklärt haben, können wir die Programmiersprache ELAN erweitern um problemspezifische Objekte, indem wir neudefinierte Datentypen, Operatoren und Prozeduren definieren und sie über Schnittstellen an nachfolgende Programmteile herausreichen. Das folgende Programmbeispiel realisiert ein Programm zur Behandlung von Problemen der ebenen Geometrie:

Beispiel:

```
PACKET ungefaehre gleichheit DEFINES aendere genauigkeit, GLEICH:
(* REAL-Zahlen werden hier für geometrische Anwendungen als gleich
   angesehen, wenn sie sich um weniger als "wenig" unterscheiden.*)

REAL VAR wenig :: 1.0E-12;

PROC aendere genauigkeit (REAL CONST wert):

IF wert <=0.0 OR wert >= 0.5
     THEN put (" wenig ausserhalb des zulaessigen Bereiches ");
          put (" auf 0.1 gesetzt ");
          wenig := 0.1
     ELSE wenig := wert
END IF
END PROC aendere genauigkeit;

BOOL OP GLEICH (REAL CONST links, rechts):
   (abs (links)* wenig + abs (rechts) * wenig) / 2.0
   >=abs (links - rechts)
   (* "underflow" wird nicht abgefangen *)
END OP GLEICH
END PACKET ungefaehre gleichheit;

PACKET punkt geometrie DEFINES PUNKT, punkt denoter, abszisse,
       ordinate, abstand, =, put, get:
TYPE PUNKT = STRUCT (REAL x, y);
   (* kartesische koordinaten x und y *)

PUNKT PROC punkt denoter (REAL CONST x,y):

   PUNKT: (x,y)

END PROC punkt denoter;

REAL PROC abszisse (PUNKT CONST p):
   p.x
END PROC abszisse;

REAL PROC ordinate (PUNKT CONST p):
   p.y
END PROC ordinate;

REAL PROC abstand (PUNKT CONST p1, p2):
   sqrt ((p1.x-p2.x) * (p1.x-p2.x) + (p1.y-p2.y) * (p1.y-p2.y))
END PROC;

BOOL OP = (PUNKT CONST p1, p2):
   abstand (p1,p2) GLEICH 0.0
END OP =;
```

```
PROC get (PUNKT VAR p):
  get (p.x); get (p.y)
END PROC get;

PROC put (PUNKT CONST p):
  put ("(" + text (p.x) + "|" + text (p.y) + ")")
END PROC put
END PACKET punkt geometrie;

PACKET geraden geometrie DEFINES GERADE, geraden denoter, senkrecht,
       steigung, ordinate, PARALLEL, =, laenge, put, get:

TYPE GERADE = STRUCT (PUNKT p1, p2);
(* auch andere Formen der Darstellung möglich, etwa Hesse'sche
   Normalform *)

GERADE PROC geraden denoter (PUNKT CONST p1, p2):
  IF p1 = p2
  THEN errorstop ("p1 = p2, Gerade nicht definiert")
  END IF;
  GERADE: (p1, p2)
END PROC geraden denoter;

BOOL PROC senkrecht (GERADE CONST g):
  abszisse (g.p1) GLEICH abszisse (g.p2)
END PROC senkrecht;

REAL PROC steigung (GERADE CONST g):
  (* liefert m aus y = mx + n *)
  IF abszisse (g.p1) GLEICH abszisse (g.p2)
  THEN errorstop ("Gerade vertikal, Steigung nicht definiert")
  END IF;
  (ordinate (g.p1) - ordinate (g.p2))/
  (abszisse (g.p1) - abszisse (g.p2))
END PROC steigung;

REAL PROC ordinate (GERADE CONST g):
  (* liefert n aus y = mx + n *)
  ordinate (g.p1) - steigung (g) * abszisse (g.p1)
END PROC ordinate;

BOOL OP PARALLEL (GERADE CONST g1, g2):
  IF genau eine senkrechte
     THEN FALSE
  ELIF beide senkrecht
     THEN TRUE
     ELSE steigungen gleich
  END IF.
  ( *hierdurch kein Abbruch bei Vertikalen * )

  genau eine senkrechte:
       senkrecht (g1) XOR senkrecht (g2).

  beide senkrecht:
       senkrecht (g1) AND senkrecht (g2).

  steigungen gleich:
       steigung (g1) GLEICH steigung (g2)
END OP PARALLEL;
```

```
BOOL OP = (GERADE CONST g1, g2):
(* Zwei Geraden sind gleich, wenn aus y =mx + n jeweils m und n
   gleich sind *)
   (g 1 PARALLEL g 2)
   AND
   (ordinate (g1) GLEICH ordinate (g2))
END OP =;

REAL PROC laenge (GERADE CONST g):
   abstand (g.p1, g.p2)
END PROC laenge;

PROC get (GERADE VAR g):
   get (g.p1); get (g.2)
END PROC get;

PROC put (GERADE CONST g):
   put (gleichung).

   gleichung:
   IF senkrecht (g)
      THEN vertikale
      ELSE standardgleichung
   END IF.

   vertikale:
   TEXT VAR vertikaltext :: "x=";
   vertikaltext CAT text (abszisse (g.p1));
   vertikaltext.

   standardgleichung:
   TEXT VAR gleichungstext ::"y =";
   gleichungstext CAT text (steigung (g));
   gleichungstext CAT "x +";
   gleichungstext CAT text (ordinate (g));
   gleichungstext
END PROC put
END PACKET geraden geometrie;

PACKET viereck geometrie DEFINES
   VIERECK, =, put, get:
TYPE VIERECK = ROW 4 PUNKT;

BOOL OP = (VIERECK CONST d1, d2):
   (* ueberprueft, ob zwei Vierecke uebereinander liegen *)

   IF erster punkt im ersten viereck gleich einem punkt im zweiten
   viereck
   THEN vergleiche restliche punkte
   ELSE viereck nicht gleich
   END IF.
```

```
   erster punkt im ersten viereck gleich einem punkt im zweiten viereck:
   INT VAR i;
   FOR i FROM 1 UPTO 4 REP
      IF d1 [1] = d2 [i]
      THEN LEAVE erster punkt im ersten viereck gleich einem punkt
                 im zweiten viereck WITH TRUE
      END IF
   END REP;
   FALSE.

   vergleiche restliche punkte:
   INT VAR j;
   FOR j FORM 2 UPTO 4 REP
      ueberpruefe naechsten punkt
   END REP;
   TRUE.

   ueberpruefe naechsten punkt:
   IF NOT (d1 [j]  = d2 [verschobener index zweites viereck])
   THEN LEAVE = WITH FALSE
   END IF.

   verschobener index zweites viereck:
   ((i+j-2) MOD 4)+ 1.

   viereck nicht gleich:
   FALSE

END OP=;

PROC get (VIERECK VAR d):
   INT VAR zaehler;
   FOR zaehler FROM 1 UPTO 4 REP
   get (d  [zaehler] )
   END REP;
END PROC get;

PROC put (VIERECK CONST d):
   INT VAR zaehler;
   put ("Viereck [");
   FOR zaehler FROM 1 UPTO 4 REP
   put (d  [zaehler] )
   END REP;
   put ("]")
END PROC put
END PACKET viereck geometrie
```

Das erste Paket "ungefähre gleichheit" definiert u.a. einen Operator "GLEICH", der bestimmt, wann zwei REAL-Werte gleich sein sollen. Der Operator "GLEICH" wird in fast allen folgenden Paketen verwendet.

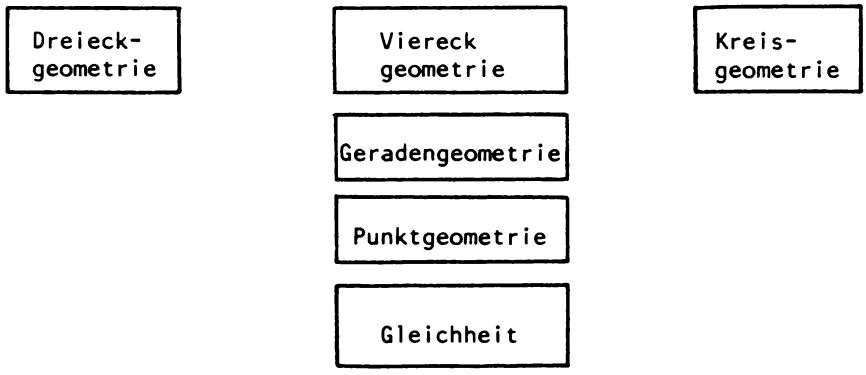
Das zweite Paket definiert Objekte zur Behandlung von Punkten. Darauf aufbauend wurde eine drittes Paket geschrieben, das Geraden behandelt. Mit Hilfe des Geraden-Pakets kann man dann Vierecke realisieren.

Diese vier Pakete stellen kein gültiges ELAN-Programm dar, denn es fehlt das "main packet". Jedoch können diese Pakete getrennt vom "main packet" und auch eines nach dem anderen - sofern man die hier angegebene Reihenfolge einhält - von einem ELAN-Compiler vorübersetzt werden. Je nach Implementation des Compilers ist es möglich, solche vorübersetzten Pakete den Standard-Paketen anzufügen und somit allen ELAN-Benutzern zugänglich zu machen; als weitere Möglichkeit kann man diese vorübersetzten Pakete nur bestimmten Benutzern oder Benutzergruppen und u.U. nur für einen bestimmten Dialog zur Verfügung stellen.

In diesem Beispiel definiert jedes Paket eine "Fachsprache". Wir reden von Fachsprachen, weil der Sprachumfang von ELAN durch problemspezifische Elemente (hier Prozeduren, Operanden und Datentypen) erweitert wird. Eine solche Fachsprache kann auch selbständig ohne Benutzung von Elementen aus ELAN, auf der sie aufbaut, benutzt werden.

Bei unserem Beispiel bauen diese vier Fachsprachen aufeinander auf, d.h. eine höhere Ebene verwendet Elemente der darunter liegenden Ebene(n).

Andererseits können Pakete konstruiert werden, die nach abstraktionsmäßigen Gesichtspunkten "parallel" liegen (vergl. die Module für Dreieck- und Kreisgeometrie, die der Einfachheit halber nicht in das Programmlisting mit aufgenommen wurden):



Da ein Programmierer somit nur in Fachsprachen programmieren kann, ohne die vorgegebenen Elemente der Programmiersprache ELAN zu verwenden, sprechen wir auch von Abstraktionsebenen. Zu einer Abstraktionsebene können ein oder mehrere - auch "parallele" - Pakete gehören. Weiterhin können Abstraktionsebenen aufeinander aufbauen. Abstraktionsebenen werden also durch die logische Struktur eines Problems bestimmt, während Pakete (Module) und darin enthaltene abstrakte Datentypen die sprachlichen Mittel der Programmiersprache ELAN sind, Abstraktionsebenen zu verwirklichen.

12. Dateien ("files")

Wie bei den meisten höheren Programmiersprachen, so existiert auch in ELAN die Möglichkeit, Daten auf externen Speichermedien in einer Datei ("file") über die Dauer der Ausführung eines Programmes hinaus aufzubewahren. In ELAN sind jedoch die Dateien losgelöst von der Art und den spezifischen Eigenschaften des Speichermediums, sowie systembedingten Eigenarten der jeweiligen Datenverarbeitungsanlagen. Durch diese Geräte- und Systemunabhängigkeit wird eine größtmögliche Vereinfachung für den Benutzer erreicht, da die notwendige Verwaltung der Dateien von ELAN in Zusammenarbeit mit dem jeweiligen Betriebssystem durchgeführt wird. Da die Realisierung von Dateien vom jeweiligen Betriebssystem der Anlage abhängt, ist die Dateibehandlung in ELAN nicht ein Teil der Sprache, sondern wird in den Standard-Paketen realisiert.
Wir unterscheiden zwischen dem definierten und dem erweiterten Standard (vergl. Kap 13.). Objekte des erweiterten Standards werden durch ⌘) gekennzeichnet.

12.1. Dateiarten

Je nach der Benutzungsart unterscheidet man in ELAN zwischen zwei standardmäßig vorgegebenen Arten von Dateien:

- Dateien im sequentiellen Zugriff ("sequential access files"), die wir kürzer als sequentielle Dateien bezeichnen. Hier sind Zugriffe (Lesen bzw. Schreiben) nur in strikter Aufeinanderfolge der Sätze oder der einzelnen Daten ("sequentiell") möglich, angefangen bei der ersten Zeile oder dem ersten Datum der Datei. Sprünge sind hier weder nach vorn noch nach hinten zulässig (Ausnahme: Rückpositionieren zum Dateianfang).

  Eine sequentielle Datei ist dann sinnvoll, wenn die gesamte Information der Datei in strikter Aufeinanderfolge bearbeitet werden soll.

- Dateien im direkten bzw. wahlfreien Zugriff ("direct access files", "random access files"), die wir kurz (jedoch etwas salopp) als direkte Dateien bezeichnen. Bei ihnen sind Zugriffe zu jedem beliebigen Satz der Datei über einen Satzschlüssel, unabhängig von der Position dieses Satzes, möglich, ohne die vorhergehenden Sätze bearbeitet zu haben. Bei vielen Implementationen ist dieser Schlüssel beispielsweise ein Integer-Wert, der eine (verborgene) Zeilennummer angibt. Ohne Kenntnis des Schlüssels ist der Zugriff auf einen Satz einer direkten Datei nicht möglich.

Diese beiden Dateiarten sind als Datentypen vorgegeben:

FILE für sequentielle Dateien,
DIRFILE für direkte Dateien.

Weitere Dateiarten können natürlich noch zusätzlich definiert werden, jedoch beschränken wir uns hier auf die Beschreibung der im Standard aufgeführten.

12.2. Deklaration und Assoziierung von Dateien

Wie schon bei den anderen Datenobjekten so ist auch bei Dateien eine Deklaration notwendig. Die geschieht durch die Angabe

- des Typs,
- des Accessattributs und
- des Namens.

Accessattribute sind auch hier wieder CONST (nur Lesezugriffe auf die Datei sind erlaubt) und VAR (Lese- und Schreibzugriffe möglich).

Der Name unterliegt den üblichen Einschränkungen, wie sie in Abschnitt 2.2. beschrieben sind.

Beispiele:

FILE CONST datei 1;
FILE VAR datei 2;
DIRFILE CONST preisliste;
DIRFILE VAR bibliothekskatalog

Um nun mit einem so deklarierten Datenobjekt arbeiten zu können, muß man es noch mit einer Bezeichnung assoziieren, anhand derer dem Betriebssystem die Zuordnung einer bestimmten physischen Datei auf einem externen Speichermedium (Magnetband, Magnetplatte etc.) möglich ist. Für die meisten Implementationen bedeutet das, daß man zunächst über die Kommandosprache des jeweiligen Betriebssystems eine Datei für den Auftrag zur Verfügung stellt, deren Name dann als Bezeichner für die Assoziierung mit dem Datenobjekt (d.h. der Datei) in ELAN verwendet wird. Diese Datei kann eine neu kreierte sein oder eine alte, die bereits mit Daten beschrieben wurde.

Mit der Assoziierung einer Datei wird diese auch gleichzeitig eröffnet. Daher existiert keine spezielle "open" Prozedur.

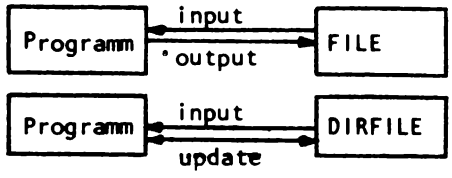
Die Assoziierung wird meistens direkt bei der Deklaration der Datei mit Hilfe einer Initialisierung durchgeführt. Sie geschieht, je nach Dateiart, mit einer der folgenden, im Standard-Paket vorhandenen Prozeduren:

FILE PROC sequential file (TRANSPUTDIRECTION CONST richtung,
TEXT CONST bezeichnung)

DIRFILE PROC direct file (TRANSPUTDIRECTION CONST richtung,
TEXT CONST bezeichnung)

Für die Betriebsrichtung einer Datei wurde der Datentyp TRANSPUTDIRECTION geschaffen. Dieser Typ wird ausschließlich in diesem Zusammenhang verwandt und besitzt nur drei Werte, die als Konstanten in den Standard-Paketen prädefiniert sind.

- input d.h. nur Leseberechtigung,
- output d.h. nur Schreibberechtigung,
- update d.h. Lese- und Schreibberechtigung.



Hierbei sind für den Typ FILE nur die Konstanten "input" und "output", für DIRFILE nur "input" und "update" zugelassen.

Beispiel:

FILE CONST datei :: sequential file (input, "datei 47");
DIRFILE VAR schuelerliste :: directfile (update, "KLASSE 12")

oder

TEXT VAR dateiname;

:

get (dateiname); (zu "get" siehe 12.4.1. *)*
FILE VAR outfile :: sequential file (output, dateiname);

:

Hinweis:

Bei den meisten Anwendungen überwiegen bestimmte Ein-/Ausgabeanweisungen für gewisse Dateien, die bestimmten Ein-/Ausgabegeräten (meist Kartenleser und Schnelldrucker) zugeordnet sind. Diese Dateien betrachten wir als Standard-Ein-/Ausgabedateien und nennen sie hier "sysin" bzw. "sysout" in Anlehnung an die Terminologie vieler Betriebssysteme. "sysin" und "sysout" sind zur Vereinfachung ihrer Dateibehandlung stets prädefiniert und assoziiert und über Prozeduren mit verkürzter (oder gänzlich ohne) Parameterliste bearbeitbar, wie wir im folgenden noch näher erläutern werden. Man beachte, daß die Zuordnung der Dateien "sysin" bzw. "sysout" zu Geräten anlagenspezifisch sind. D.h. es kann sich bei einer Anlage um den Kartenleser und Drucker, bei einer anderen jedoch um ein zugeordnetes Dialogterminal handeln.

12.3. Verwaltung von Dateien

Die Verwaltung von Dateien erfolgt mit Prozeduren, die diese schließen oder löschen.

Wie schon im vorigen Abschnitt erwähnt, existiert keine eigentliche "open"-Prozedur. Das Eröffnen von Dateien wird implizit durch deren Assoziierung vorgenommen. Zum Schließen von Dateien gibt es die Prozeduren

PROC close (FILE CONST datei)

und

PROC close (DIRFILE CONST datei)

Alle benutzten Dateien sollten spätestens bei Programmende explizit geschlossen werden, jedoch werden aus Sicherheitserwägungen zum Programmende auch alle nicht vom Benutzer geschlossenen Dateien implizit geschlossen.

Das Löschen von Dateien geschieht durch die Prozeduren

PROC erase (FILE CONST datei)

bzw.

PROC erase (DIRFILE CONST datei)

12.4. Ein-/Ausgabe

Im ELAN-Standard stehen zur Bearbeitung von Dateien Prozeduren zur Verfügung. Sie unterteilen sich in

- datenobjektorientierte Ein-/Ausgabeprozeduren;
  sie beziehen sich nur auf einzelne Datenobjekte der vier einfachen Datentypen in sequentiellen Dateien;
- zeilenorientierte Ein-/Ausgabeprozeduren;
  sie sind für direkte Dateien die einzig möglichen Lese- und Schreibzugriffe, jedoch auch für sequentielle Dateien verwendbar;
- Zeilenverwaltung;
  hierdurch werden explizite Zeilen- und Seitenvorschübe bewirkt;
- Informationsprozeduren.

12.4.1. Datenobjektorientierte Eingabe

Für die einfache datenobjektorientierte Eingabe über "sysin" gibt es für die einfachen Datentypen folgende Eingabeprozeduren:

PROC get (TEXT VAR string)
PROC get (INT VAR zahl)
PROC get (REAL VAR zahl)
PROC get (BOOL VAR wert)

Diese Prozeduren lesen jeweils den ersten oder nächsten Denoter von dem Standard-Eingabemedium ("sysin") und überprüfen dann, ob der gelesene Typ mit dem des verwendeten aktuellen Parameters übereinstimmt. Dann wird dem aktuellen Parameter der eingelesene Wert zugewiesen. Im Fall ungleicher Typen wird eine Beendigung des Programmlaufs mit Fehlermeldungen durchgeführt.

Es wird also von der Prozedur der richtige Typ erwartet und nicht etwa solange gelesen, bis endlich ein Denoter des passenden Typs gefunden wird.

Der Versuch, über das Dateiende hinaus zu lesen, führt zum Programmabbruch mit Fehlermeldung. Dies gilt auch für alle anderen Eingabeprozeduren und wird dort nicht noch jeweils extra erwähnt. Die Abfrage, ob das Ende der

Datei erreicht ist, wird in den Informationsprozeduren in Abschnitt 12.4.5. erläutert. Bei einer Dialog-Ein-/Ausgabe (z.B. Fernschreiber oder Sichtgerät als "sysin") ist eine solche Abfrage auf Dateiende (außer bei Dialogende) selbstverständlich nicht möglich. Hier führt jedes "get" dazu, daß das Dialoggerät in den Eingabemodus geht.

Ferner ist zu beachten:

- Die Datei "sysin" ist bei jedem Programmlauf standardmäßig vorhanden und darf nicht nochmals durch den Benutzer definiert werden.
- Zu lesende Denoter müssen durch mindestens ein Leerzeichen getrennt werden; das Zeilenende gilt gleichfalls als Begrenzung.
- Zu lesende TEXT-Objekte werden im Gegensatz zu ihrer Denoter-Darstellung (3.3.d) ohne einschließende Anführungszeichen geschrieben. Auch hier gilt bereits ein Blank (bzw. Zeilenende) als Trennzeichen zwischen den einzelnen TEXT-Objekten.

Beispiel:

Hamburg-Amerika-Doppelschraubendampferluxuskabinenschlüssellochbesen

ist 1 TEXT-Objekt

Hamburg Amerika Doppelschrauben Dampfer Luxus Kabinen Schlüssel Loch Besen

sind 9 TEXT-Objekte. Das Objekt

"DDR"

wird mitsamt der einschließenden Anführungszeichen gelesen und abgespeichert.

Beispiel:

```
TEXT VAR name, vorname;
BOOL VAR person ist maennlich;
INT  VAR geburtsjahr;
REAL VAR einkommen;
:
:
get (name);
get (vorname);
get (person ist maennlich);
get (geburtsjahr);
get (einkommen);
:
:
(* Programmende *)
(* Daten:       *)
Duck Dagobert TRUE 1929 1.0E15
```

Betriebssysteme verlangen nach einer Kennzeichnung, die den Programmtext von den Eingabedaten trennt. Diese Trennung wird in unseren Beispielen durch die Kommentarzeilen (* *Programmende* *) und (*Daten:*) angedeutet.

Da gerade beim Einlesen von TEXT die Trennung der Eingabe durch ein Leerzeichen manchmal unerwünscht ist, gibt es noch zwei weitere "get"-Prozeduren:

PROC get (TEXT VAR string, INT CONST maxlaenge)

Hiermit wird ein TEXT-Objekt mit der maximalen Länge "maxlänge" von der aktuellen Leseposition eingelesen. Die maximale Länge kann nicht voll ausgeschöpft werden, wenn die restliche Zeile (aktuelle Leseposition bis Zeilenende) kürzer ist. Es wird dann nur die vorhandene Restzeile eingelesen. Die aktuelle Leseposition ist entweder der Dateianfang (beim ersten Lesebefehl oder nach Rückpositionieren, siehe 12.4.4.), der Anfang einer neuen Zeile (falls durch den letzten durchgeführten Lesebefehl die alte Zeile ausgeschöpft war) oder das Zeichen hinter dem Trennzeichen bzw. das Zeichen, das auf den Text folgt, der mit der Längenbegrenzung "maxlänge" durch die obige Prozedur gelesen wurde.

PROC get (TEXT VAR string, TEXT CONST begrenzer)

Es wird ein TEXT-Objekt gelesen, das durch den Parameter "begrenzer" abgeschlossen wird, jedoch längstenfalls bis zum Zeilenende.

Beispiel:

```
(* liest die Zeilen des Gedichtes "Ein Gleiches" von J.W.Goethe (1749-
1832) nacheinander einzeln ein  *)

ROW 8 TEXT VAR zeile;
TEXT CONST ende :: "STOP";
INT CONST laenge :: 17;
:
:
get (zeile [1], ende);
get (zeile [2], 9);
get (zeile [3], laenge);
get (zeile [4], laenge - 5);
skip restzeile;
get (zeile [5], "*");
get (zeile [6], "**");
skip restzeile;
get (zeile [7], laenge);
get (zeile [8], 15),

skip restzeile:
   (* soll die restlichen Zeichen der aktuellen Zeile überlesen; nicht
      naeher definiert *)
:
:
(* Programmende *)
(* Daten:       *)

Ueber allen GipfelnSTOPIst Ruh, In allen Wipfeln Spuerest du
Kaum einen Hauch;*Die Voeglein schweigen im Walde.**
Warte nur, balde Ruhest du auch.
```

Den "get"-Aufruf entsprächen in diesem Beispiel die folgenden Zuweisungen:

```
zeile [1] := "Ueber allen Gipfeln";
zeile [2] := "Ist Ruh, ";
zeile [3] := "In allen Wipfeln ";
zeile [4] := "Spuerest du ";
zeile [5] := "Kaum einen Hauch;";
zeile [6] := "Die Voeglein schweigen im Walde.";
zeile [7] := "Warte nur, balde ";
zeile [8] := "Ruhest du auch."
```

Schließlich existiert noch

```
TEXT PROC get
```

Sie ist parameterfrei, liest also nicht in einen aktuellen Parameter einen Wert, sondern liefert, im Gegensatz zu den übrigen "get"-Prozeduren, selbst einen Wert vom Typ TEXT. Ihre Anwendung liegt hauptsächlich in der Initialisierung von CONST-Datenobjekten durch Einlesen eines Wertes, die ja mit den bisher beschriebenen "get"-Prozeduren nicht möglich war.

Will man mit dieser Prozedur andere als TEXT-Konstanten initialisieren, so muß man das gelesene TEXT-Objekt mit der entsprechenden Konvertierungsroutine (Konvertierungen siehe 13.4.) in den richtigen Typ umwandeln.

Beispiel:

```
TEXT CONST ueberschrift :: get;
INT  CONST vergroesserungsfaktor :: int (get);
REAL CONST mehrwertsteuersatz :: real (get);

(* int (...) bzw. real (...) sind Konvertierungsprozeduren, die einen TEXT
   in INT bzw. REAL umwandeln *)

:
:
(* Programmende *)

(* Daten: *)

Kosten-Nutzen-Analyse  15  0.13
```

Für andere sequentielle Dateien als "sysin" existieren noch zusätzlich die folgenden, entsprechenden Eingabeprozeduren:

```
PROC get (FILE CONST f, TEXT VAR string)
PROC get (FILE CONST f, INT  VAR zahl)
PROC get (FILE CONST f, REAL VAR zahl)
PROC get (FILE CONST f, BOOL VAR wert) :)

PROC get (FILE CONST f, TEXT VAR string, TEXT CONST begrenzer)
PROC get (FILE CONST f, TEXT VAR string, INT  CONST maxlaenge)
TEXT PROC get (FILE CONST f)
```

Diese Prozeduren arbeiten analog den zuvor erwähnten, jedoch statt auf "sysin" auf einer zuvor deklarierten sequentiellen Datei.

Beispiel:

```
TEXT VAR artikel
INT  VAR anzahl, artikelnr;
REAL VAR preis;
FILE CONST artikelliste :: sequential file (input, "ADAT-VS");
:
:
get (artikelliste, artikelnr);
get (artikelliste, artikel);
get (artikelliste, anzahl);
get (artikelliste, preis);
:
:
```

12.4.2. Datenobjektorientierte Ausgabe

Zur Ausgabe einzelner Datenobjekte auf "sysout" stehen die folgenden Ausgabeprozeduren zur Verfügung:

```
PROC put (TEXT CONST string)
PROC put (INT  CONST zahl)
PROC put (REAL CONST zahl)
PROC put (BOOL CONST wert)
```

Da die meisten Ausgabemedien zeilenorientiert sind und nicht Zeichen oder Teile einer Zeile ausgeben können, ist es sinnvoll, von ELAN einen impliziten Zeilenpuffer vorzugeben. Die obigen Prozeduren setzen die zu schreibenden Werte in diesen Zeilenpuffer ab, sofern noch genügend Platz vorhanden ist; andernfalls wird der Puffer erst an "sysout" ausgegeben, gelöscht und dann mit den Werten aufgefüllt (Abweichung für TEXT siehe weiter unten). Außerdem wird der Zeilenpuffer ausgegeben, wenn

- ein expliziter Zeilen- oder Seitenvorschub verlangt wird (siehe "Zeilenverwaltung", 12.4.4.),
- sich die Betriebsrichtung ändert, d.h. ein "get" von "sysin" gemacht wird,
- das Programm beendet wird.

Positive INT- und REAL-Objekte werden mit einer führenden Leerstelle anstatt des positiven Vorzeichens dargestellt.

Befinden sich mehrere Objekte im Zeilenpuffer, so werden sie voneinander durch je ein Leerzeichen getrennt (speziell für positive INT- und REAL-Objekte bedeutet dies, daß durch diese Trennung zwei Leerstellen entstehen). Somit erreicht man, daß durch die Trennung die Ausgabe zweier "put"-Anwendungen auch als separate Objekte erkannt werden kann.

Im Kontext des vorherigen Beispiels ergibt sich folgende Ausgabe:

```
zahl = 12 TRUE 5.0E-1
```

Hierbei fallen durch die Konkatenation mittels des Operators "&" die trennenden Leerzeichen weg, die bei separater Ausgabe (hier z.B. durch vier "put"-Anwendungen) in den Zeilenpuffer eingefügt werden.

Wie schon bei den Eingabeprozeduren so sind auch die Ausgabeprozeduren für andere sequentielle Dateien erklärt:

```
PROC put (FILE CONST f, TEXT CONST string)
PROC put (FILE CONST f, INT  CONST zahl)
PROC put (FILE CONST f, REAL CONST zahl)
PROC put (FILE CONST f, BOOL CONST wert) :*)
```

Ihre Wirkungsweise entspricht den vorangegangenen Ausgabeprozeduren.

Beispiel:

```
FILE VAR dat    :: sequential file (output, "XYZ");
TEXT VAR wort;
INT  VAR zahl;
REAL VAR bruch  :: 0.5;
:
:
wort  :=  "hanni";
zahl  :=  100156;
put  (dat, wort + "ball");
put  (dat, bruch ** 3);
put  (dat, zahl);
put  (dat, zahl < 100165);
:
:
```

Daten in der Datei:

```
hanniball 1.25E-1 100156 TRUE
```

12.4.3. Zeilenorientierte Ein-/Ausgabe

Die im vorangegangenen Abschnitt besprochenen Ein-/Ausgabeprozeduren boten keine Möglichkeit, auf eine direkte Datei zuzugreifen. Diese Dateiart verlangt auch schon von ihrer Struktur her eine andere als datenobjektorientierte Behandlung. Hier sind die ansprechbaren Datenobjekte die einzelnen Sätze der Datei, die über einen Schlüssel adressiert werden.

Die Leseprozedur ist vereinbart als

```
PROC getline (DIRFILE CONST f, TEXT CONST schluessel, TEXT VAR satz)
```

Im definierten Standard. Abweichend gilt für viele Implementationen, daß der Satzschlüssel "schlüssel" nicht ein TEXT CONST-, sondern ein INT CONST-Datenobjekt, nämlich die Satznummer in der Datei ist.

Paßt ein TEXT nicht mehr vollständig in eine restliche Zeile, so wird er, abweichend von den anderen einfachen Datentypen - laut definiertem Standard - so weit wie möglich dort abgesetzt, und nach Ausgabe des Puffers wird der Rest des TEXT-Objektes an den Anfang des Puffers geschrieben. Im Gegensatz hierzu verfahren die meisten Implementationen jedoch wie bei INT, REAL und BOOL ohne diesen Umbruch.

Beispiel:

```
TEXT VAR   wort  :: "Komm";
INT  VAR   zahl  :: 12;
REAL CONST bruch :: 0.5;
:
:
put (zahl);
put (zahl + 3);
put (18);
put ("und jetzt der Bruch:");
put (bruch);
get (zahl);    (* bewirkt u.a. einen Zeilenwechsel in der Ausgabe *)
put ( 0 < zahl AND zahl < 5);
put (wort + "puter");
:
:
(* Programmende *)
(* Daten:       *)
2
```

Ausgabe:

```
 12  15  18 und jetzt der Bruch:  5.0E-1
TRUE Kommputer
```

Wie man an dem Beispiel sieht, sind selbstverständlich auch Ausdrücke einschließlich Denotern und CONST-Objekten als aktuelle Parameter zugelassen.

Im erweiterten Standard (vergl. Kap. 13) ist noch der Operator "&" als dyadischer Operator zwischen allen Kombinationen von TEXT, INT, REAL und BOOL definiert. Er wandelt die Operanden in TEXT um - sofern erforderlich - und konkateniert sie. Dies erleichtert in vielen Fällen die Ausgabe.

Beispiel:

```
:
:
put ("zahl =" & zahl);
put ((bruch < 2.5) & bruch);
:
:
```

Die Prozedur liest aus der Datei "f" den Satz mit dem Schlüssel "schlüssel" in den Parameter "satz".

Soll ein Satz gelesen werden, dessen Satzschlüssel nicht existiert, so liefert die Prozedur einen niltext an "satz".

Der Versuch, einen zu langen Satz (man beachte die maximale Länge eines TEXT-Datenobjekts) zu lesen, führt zum Programmabbruch mit Fehlermeldung.

Darüberhinaus sind noch die folgend drei "getline" vereinbart:

PROC getline (DIRFILE CONST f, TEXT VAR satz) ¤)

liest sequentiell Sätze aus einer direkten Datei (sofern diese einen Schlüssel vom Typ INT hat);
entsprechend für sequentielle Dateien:

PROC getline (FILE CONST f, TEXT VAR satz)

und zum Lesen eines Satzes von "sysin"

PROC getline (TEXT VAR satz) ¤)

Beispiel:

```
DIRFILE CONST d datei  ::  direkt file (input, "DXYZ");
FILE    CONST s datei  ::  sequential file (input, "SXYZ");
TEXT VAR zeile;
TEXT VAR satzschluessel;
:
satzschluessel := "10";
getline (d datei, satzschluessel, zeile);
getline (d datei, "12", zeile);
:
getline (s datei, zeile);
:
getline (zeile);
:
```

Die entsprechende Schreibprozedur im definierten Standard ist

PROC putline (DIRFILE CONST f, TEXT CONST schluessel, satz)

Auch hier weichen viele Implementationen ab: statt mit TEXT CONST ist "schlüssel" wieder als INT CONST vereinbart.

Die Prozedur schreibt den Text "satz" in die direkte Datei "f" und zwar in den Satz, der mit "schlüssel" angesprochen wird.

PROC putline (DIRFILE CONST f, TEXT CONST) *)

schreibt sequentiell den Text "satz" in den nächsten Satz der direkten Datei "f"; entsprechend verfährt

PROC putline (FILE CONST f, TEXT CONST satz)

für sequentielle Dateien und

PROC putline (TEXT CONST satz) *)

Ist bei diesen Prozeduren der auszugebende Satz länger als die maximal zulässige Zeilenlänge der angesprochenen Datei, so wird der Satz (ohne Warnung) entsprechend verkürzt ausgegeben. Die maximal zulässige Zeilenlänge läßt sich mit der Informationsprozedur "maxlinelength (f)" erfragen (s. auch 12.4.5.).

Beispiel:

```
FILE VAR s datei :: sequential file (output, "SXYZ");
DIRFILE VAR d datei :: derect file  (update, "DXYZ");
TEXT  VAR satznr :: "B",
          satz   ::
      "Wer ELAN nicht ehrt, ist den Computer nicht wert.";
:
:
putline (s datei, satz);
putline (d datei, "ELAN am Abend,erquickend und labend.", satznr);
putline (d datei, satz, "A");
putline ("Das war's.")
```

12.4.4. Zeilenverwaltung

Für einen expliziten Zeilenwechsel hat man folgende Prozeduren zur Verfügung:

PROC line (FILE CONST f)
PROC line

PROC line (FILE CONST f, INT CONST anzahl)
PROC line (INT CONST anzahl)

Die ersten beiden Prozeduren geben den Inhalt des Zeilenpuffers (ggfs. niltext bei leerem Puffer) in die sequentielle Datei "f" bzw. im zweiten Fall auf "sysout" aus und löschen den Zeilenpuffer, sofern die Betriebsrichtung "output" ist. Durch die Angabe von der Integer "anzahl" wird dieser Vorgang entsprechend oft wiederholt.

Für sequentielle Dateien, die eine Seitenstruktur besitzen, wie z.B. der Drucker (als "sysout"), gibt es außerdem noch die Möglichkeit, nicht nur auf die nächste Zeile, sondern auch auf die nächste Seite zu positionieren mit den Prozeduren

PROC page (FILE CONST f)
PROC page

Auch hier wird zuerst der Zeilenpuffer auf die Datei "f" bzw. "sysout" (im parameterlosen Fall) ausgegeben und dann gelöscht. Die nächste Ausgabe des Puffers erfolgt auf einer neuen Seite.

Beispiel:

```
FILE VAR textdatei  :: (output, "BfV-VS")
INT CONST dreimal   :: 3;
:
:
put (textdatei, "Jetzt machen wir mal 'nen Zeilenwechsel");
line (textdatei);
put  (textdatei, "Noch dreimal,");
line (textdatei, dreimal);
putline (textdatei, "weil's Spass macht");
line (textdatei);
put  (textdatei, "Schluss der Spielerei.").
:
:
```

Daten in "textdatei":

```
Jetzt machen wir mal 'nen Zeilenwechsel
Noch dreimal,


weil's Spass macht

Schluß der Spielerei.
```

Um in einer sequentiellen Datei wieder an den Anfang zu gelangen (z.B. wenn man die Daten noch ein weiteres Mal lesen will), kann man mit der Prozedur

PROC reset (FILE CONST f)

auf den Anfang der Datei "f" zurückpositionieren. D.h. der nächste zu bearbeitende Satz der Datei ist der erste der Datei in der sequentiellen Abfolge.

Zusätzlich sind zum Löschen einzelner oder auch mehrerer Zeilen einer direkten Datei entsprechende Prozeduren in den erweiterten Standard aufgenommen worden.

PROC delete (DIRFILE CONST f, INT CONST schluessel) *)

löscht die Zeile mit der Zeilennummer "schluessel" aus der Datei "f".

PROC delete (DIRFILE CONST f, INT CONST von, bis) *)

löscht aus der Datei "f" die Zeilen die zwischen "von" und "bis" (einschließlich der Grenzen) liegen. Diese letzte Prozedur ist nur für solche Implementationen sinnvoll, bei denen der Satzschlüssel ein Integer-Wert ist.

12.4.5. Informationsprozeduren

Wie schon bei den Ausgabeprozeduren erwähnt, ist es häufig nützlich, das Dateiende rechtzeitig zu erkennen. Die Prozeduren zum Abfragen des Dateiendes sind

BOOL PROC eof (FILE CONST f) für sequentielle Dateien,
BOOL PROC eof für "sysin"

Zusätzlich gibt es im erweiterten Standard, falls dieser auch sequentielle Bearbeitung von direkten Dateien mit INT-Schlüsseln vorsieht, auch noch

BOOL PROC eof (DIRFILE CONST f) *)

Bei Dateien mit Seitenstruktur kann das Überschreiten des Seitenendes mit einer der folgenden Prozeduren erfragt werden:

BOOL PROC eop (FILE CONST f) für sequentielle Dateien,
BOOL PROC eop für "sysin"

Bei der Errichtung einer Datei kann man meistens die Zeilenlänge festlegen; bei bestimmten Dateien ist die Zeilenlänge auch durch ein gewisses Ausgabemedium (z.B. Schnelldrucker) begrenzt, Diese Länge (d.h. die Anzahl der Zeichen pro Zeile) läßt sich mit den Prozeduren

INT PROC maxlinelength (FILE CONST f)
INT PROC maxlinelength (DIRFILE CONST f)
INT PROC maxlinelength

feststellen.

Besitzt die Datei eine Seitenstruktur, so erhält man die maximale Anzahl von Zeilen pro Seite durch die Prozeduren

INT PROC maxpagelength (FILE CONST f)
INT PROC maxpagelength

Ob eine Datei korrekt eröffnet wurde, kann man mit den Prozeduren

BOOL PROC opened (FILE CONST f)
BOOL PROC opened (DIRFILE CONST f)

abfragen.

Zusätzlich steht dem Benutzer des erweiterten Standards noch die Abfrage auf die Nummer der letzten bearbeiteten Zeile zur Verfügung durch die Prozeduren

INT PROC line nr (FILE CONST f) *)
INT PROC line nr (DIRFILE CONST f) *)

Ob eine Datei neu eingerichtet wurde, kann mit den Prozeduren

BOOL PROC new (FILE CONST f)
BOOL PROC new (DIRFILE CONST f)

erfragt werden. Aber Vorsicht: die Realisierung dieser Informationsprozeduren ist bei manchem Betriebssystem nicht möglich. Diese beiden Prozeduren sind daher nicht in jeder Implementation vorhanden.

13. Die Standard-Pakete

Üblicherweise gehören zu jeder Programmiersprache Routinen, insbesondere die Ein-/Ausgabe, die sehr stark an bestimmte Rechnerkonfigurationen gebunden ist. Ferner existieren einige Operationen (z.B. mathematische Routinen und Textverarbeitung), die das Programmieren für den Anwender erleichtern sollen. Da die Rechnerkonfigurationen und Anwendungsgebiete sehr unterschiedlich sind, ist es sinnvoll, all diese Operationen nicht in die Definition der Sprache aufzunehmen.

Solche Operationen können in den meisten Fällen in ELAN selbst formuliert und in Form von (vorübersetzten) Paketen zur Verfügung gestellt werden, was jedoch installationsabhängig ist. Um trotzdem eine möglichst große Übertragbarkeit von Programmen zu erreichen, wird in der Sprachbeschreibung ein Satz von Datentypen, Operationen (d.h. Prozeduren und Operatoren) und Konstanten empfohlen. Diese werden in ELAN Standard-Pakete genannt und stellen einen minimalen Standard dar, der von jeder Implementation erweitert werden kann.

Der in der Sprachbeschreibung empfohlene Standard wird auch als definierter Standard bezeichnet, im Gegensatz zu den Erweiterungen, die in vielen Implementationen vorhanden sind und sich als sinnvoll herausgestellt haben, die wir hier den erweiterten Standard nennen. Diese Erweiterungen werden in diesem Buch durch das Zeichen *) gekennzeichnet.

Einige Operationen sind sowohl als Prozedur als auch als Operator vorhanden, da einige Benutzer die operationelle Schreibweise, andere Prozeduraufrufe bevorzugen.

13.1. Standard der einfachen Datentypen

Die Operationen auf den einfachen Datentypen werden zwar - auch aus Effizienzgründen - zum größten Teil vom ELAN-Compiler realisiert, doch ist es vom Sprachkonzept her durchaus denkbar, daß auch beispielsweise die REAL-Arithmetik, die auf gewissen Kleinrechnern nicht vorhanden ist, durch ein in ELAN geschriebenes Standard-Paket realisiert wird. Dennoch verzichten wir darauf, in der folgenden Aufzählung die vier Grundrechenarten und Vorzeichen für INT und REAL zu erwähnen, ebenso wie auf die Boolschen Verknüpfungen. Die Operatoren wurden mit ihrem Operanden- und Resultatstypen bereits in den Abschnitten 4.2.1. und 4.2.2. aufgeführt.

13.1.1. Datentyp INT

INT CONST maxint

gibt den größten Integerwert der jeweiligen Implementation an.

INT CONST minint *)

entsprechend für den kleinsten Integerwert.

INT OP MOD (INT CONST a, b)

Modulofunktion; sie liefert den Rest der ganzzahligen Division aus a DIV b.

*INT OP ** (INT CONST a, b)*

Exponentation (a hoch b); b >= 0; falls a = 0, muß b > 0 sein.

OP INCR (INT VAR a, INT CONST b)

erhöht den Wert von a um b; entspricht der Zuweisung a := a + b;
liefert keinen Wert.

OP DECR (INT VAR a, INT CONST b)

vermindert den Wert von a um b; entspricht der Zuweisung a := a - b;
liefert keinen Wert.

INT OP SIGN (INT CONST a)

Signumfunktion; liefert + 1 für a > 0, 0 für a = 0 und -1 für a < 0.

INT PROC sign (INT CONST a)

Signumfunktion; wie oben.

INT OP ABS (INT CONST a)

Absolutbetrag

INT PROC abs (INT CONST a)

Absolutbetrag

INT PROC max (INT CONST a, b) *)

liefert das Maximum von a und b.

INT PROC min (INT CONST a, b) *)

liefert das Minimum von a und b.

13.1.2. Datentyp REAL

Da der Wertebereich und die Genauigkeit von REAL-Werten je nach Rechnersystem verschieden sind, existieren im Standard die entsprechenden implementationsabhängigen Konstanten:

REAL CONST maxreal

größter REAL-Wert der Implementation

REAL CONST smallreal

kleinster REAL-Wert der Implementation, so daß gilt:
1.0 + smallreal > 1.0. und 1.0 - smallreal < 1.0

REAL CONST smallestreal *)

kleinster echt positiver REAL-Wert der Implementation.
Ferner existieren die zu den im vorigen Abschnitt über INT-Datenobjekte deklarierten Operationen hier auch analog für REAL-Datenobjekte:

REAL OP MOD (REAL CONST a, b)

Modulfunktion (auf REAL erweitert); sie liefert den "Rest" der Division von a durch b (entspricht: *(a/b - floor (a/b))*b*)

Beispiel:

```
REAL CONST x:: 1.5;
put (0.5 MOD x);
put (1.0 MOD x);
put (1.5 MOD x);
put (2.0 MOD x)
```

Ausgabe:

```
5.0E-1   1.0E0   0.0E0   5.0E-1
```

*REAL OP ** (REAL CONST a, INT CONST b)*

Exponentiation (a hoch b); b >= 0; falls a = 0.0, muß b > 0 sein.

*REAL OP ** (REAL CONST a, b)* *)

Exponentiation (a hoch b); a >= 0.0; falls a =0.0, muß b > 0.0 sein; ansonsten ist b beliebig.

OP INCR (REAL VAR a, REAL CONST b)

erhöht den Wert von a um b; entspricht der Zuweisung a := a + b; liefert keinen Wert.

OP DECR (REAL VAR a, REAL CONST b)

vermindert den Wert von a um b; entspricht der Zuweisung a := a - b; liefert keinen Wert.

INT OP SIGN (REAL CONST a)

Signumfunktion; liefert die INT-Werte + 1 für a > 0.0,
0 für a = 0.0 und -1 für a < 0.0.

INT PROC sign (REAL CONST a)

Signumfunktion; wie oben.

REAL OP ABS (REAL CONST a)

Absolutbetrag

REAL PROC abs (REAL CONST a)

Absolutbetrag

REAL PROC max (REAL CONST a, b) *)

liefert das Maximum von a und b

REAL PROC min (REAL CONST a,b) *)

liefert das Minimum von a und b.

Zur Überführung von INT-Werten nach REAL und umgekehrt gibt es die folgenden Prozeduren:

REAL PROC real (INT CONST a)

wandelt den INT-Wert von a in REAL um.

Beispiel:

```
INT CONST n :: 2;
put (real (n));
put (n)
```

Ausgabe:

```
 2.0E0  2
```

INT PROC trunc (REAL CONST a)

wandelt den REAL-Wert von a in INT um durch Abschneiden der Dezimalstellen; hierbei gilt: trunc (-a) = - trunc (a)

INT PROC int (REAL CONST a) *)

arbeitet wie "trunc".

INT PROC round (REAL CONST a)

wandelt den REAL-Wert von a in INT um durch Runden; hierbei gilt: round (-a) = - round (a)

Beispiel für "trunc" und "round":

```
REAL VAR x :: 1.0;
REP
  put ("trunc (");
  put (x);
  put (") =");
  put (trunc (x));
  put ( "round (");
  put (x);
  put (") =");
  put (round (x));
  line;
  x INCR 0.5
UNTIL x >  2.0 END REP
```

Ausgabe:

```
trunc (  1.0E0 ) =  1   round (  1.0E0 ) =  1
trunc (  1.5E0 ) =  1   round (  1.5E0 ) =  2
trunc (  2.0E0 ) =  2   round (  2.0E0 ) =  2
```

13.1.3. Datentyp BOOL

Außer den Boolschen Verknüpfungsoperatoren (NOT, AND, OR, XOR), die schon in den Abschnitten 4.2.1. und 4.2.2. beschrieben wurden, sind hier noch die CONST-Datenobjekte für die beiden BOOL-Denoter deklariert.

```
BOOL CONST true   :: TRUE
BOOL CONST false  :: FALSE
```

13.1.4. Datentyp TEXT

Wie auch schon die zulässigen Wertebereiche von INT und REAL je nach Implementation beschränkt waren, kann auch die maximale Länge eines TEXT-Datenobjekts, d.h. die maximale Anzahl der Zeichen dieses Objekts begrenzt sein. Zur Zeit sind folgende maximale Längen je nach Implementation im Gebrauch: 80, 255 und 65535. Die Obergrenze kann man durch folgende Konstanten erhalten:

```
INT CONST maxtextlength   *)
```

Die Denoter für einen leeren Text (Text der Länge 0), auch "niltext" genannt, für einen Text, der aus einem Leerzeichen ("blank") besteht, und für einen Text, der aus einem Anführungszeichen (engl."quote") besteht, sind im TEXT-Standard als CONST-Objekte deklariert.

```
TEXT CONST niltext :: "",
           blank   :: " ",
           quote   :: """"
```

Die Boolschen Vergleichsoperatoren =, <> , < ,<= ,> ,> = sind auch zwischen zwei Texten deklariert. Die Ordnung von Texten wird in der "Beschreibung der Programmiersprache ELAN" wie folgt festgelegt:

"Texte werden nach den üblichen lexikalischen Regeln verglichen, wobei die Relation zwischen den einzelnen Zeichen durch die interne Codierung des benutzen Rechners oder des Compilers festgelegt ist. Rechnerunabhängig ist jedoch die Ordnung der Buchstaben a...b und A...Z und die Ordnung der Ziffern 0...9 festgelegt, wobei sich keine anderen Zeichen in den jeweiligen Reihen befinden sollen. Beim Vergleich von Texten unterschiedlicher Länge, wobei der eine Text den anderen als Anfang enthält, gilt:

a + x > a (a,x: TEXT; LENGTH x > 0)"

Weiterhin gibt es die folgenden Operationen auf Texten:

INT OP LENGTH (TEXT CONST t)

liefert die Länge eines Texts, d.h. die Anzahl der Zeichen; die Länge von "niltext" ist 0.

Beispiel:

```
TEXT CONST wort  :: "ELAN",
           satz  :: "Dieser Satz hat 27 Zeichen.";
put  (LENGTH wort);
put  (LENGTH satz);
put  (LENGTH "1")
```

Ausgabe:

```
4  27  1
```

INT PROC length (TEXT CONST t)

wie oben.

TEXT OP + (TEXT CONST a, b)

liefert einen Text, der durch Aneinanderhängen (Konkatenation) von a und b entsteht.

Beispiel:

```
TEXT CONST  vorsilbe  :: "Ent",
            endsilbe  :: "ung";
put  (vorsilbe + "steh" + endsilbe);
put  (vorsilbe + "en")
```

Ausgabe:

```
Entstehung Enten
```

OP CAT (TEXT VAR a, TEXT CONST b)

hängt den Text b an Text a; entspricht a := a + b; der Operator liefert <u>keinen</u> Wert.

*TEXT OP * (INT CONST i, TEXT CONST t)*

liefert einen Text, der durch (i-1)fache Konkatenation des Textes t mit sich selbst entsteht; mit anderen Worten: i-viele gleiche Texte t werden aneinander gehängt; i > 0.

Beispiel:

```
TEXT CONST horizontal :: "-",
           vertikal   :: "|",
           kreuzung   :: "+";
INT VAR kastenreihe, zwischenzeile;

FOR kastenreihe FROM 1 UPTO 6
REP
   erstelle kastenreihe
END REP;
zeichne trennzeile.

erstelle kastenreihe:
   zeichne trennzeile;
   FOR zwischenzeile FROM 1 UPTO 2
   REP
      zeichne zwischenzeile
   END REP.

zeichne trennzeile:
   put (6 * (kreuzung + 4 * horizontal) + kreuzung);
   line.

zeichne zwischenzeile:
   put (6 * (vertikal + 4 * blank) + vertikal);
   line
```

Ausgabe:

```
+----+----+----+----+----+----+
|    |    |    |    |    |    |
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
|    |    |    |    |    |    |
+----+----+----+----+----+----+
|    |    |    |    |    |    |
|    |    |    |    |    |    |
+----+----+----+----+----+----+
```

Man beachte in diesem Beispiel, daß auch für TEXT-Operatoren * eine höhere Priorität als + besitzt (vergl. 10.2.3.).

TEXT PROC compress (TEXT CONST t)

liefert den Text t ohne führende und nachfolgende blanks.

Beispiel:

```
TEXT CONST t  ::  "   a b c   ";
put (compress (t))
```

Ausgabe:

```
a b c
```

TEXT PROC text (TEXT CONST t, INT CONST l)

liefert einen Text aus t mit der Länge *l*, *l* > 0; ist *l* > LENGTH t, so wird rechts mit blanks aufgefüllt; ist *l* < LENGTH t, so wird t auf die angegebenen Länge (am rechten Ende) abgeschnitten.

In den folgenden Operationen bedeutet die Position p das p-te Zeichen eines Texts, von links nach rechts gezählt, beginnend beim äußerst linken Zeichen mit 1.

TEXT PROC text (TEXT CONST t, INT CONST l, p)

liefert einen Text aus t mit der Länge *l*, beginnend bei der Position p in t; es muß gelten: *l* > 0, p <= LENGTH t; falls *l* > (LENGTH t) - p + 1, wird der Text mit blanks aufgefüllt; ist *l* < (LENGTH t) - p + 1, so wird der Text abgeschnitten.

Beispiel für die beiden "text" Prozeduren:

```
TEXT CONST wort  :: "Genesender";
putline (text (wort, 4));
putline (text (wort, 4, 6));
putline (text (wort, 12)
```

Ausgabe:

```
Gene
ende
Genesender
```

TEXT PROC subtext (TEXT CONST t, INT CONST von)

liefert einen Teiltext von t, beginnend bei der Position "von" bis zum Ende des Texts; es muß gelten: 1 <= von <= LENGTH t; entspricht *text (t, LENGTH t - von + 1, von)*.

TEXT PROC subtext (TEXT CONST t, INT CONST von, bis)

liefert einen Teiltext von t, beginnend bei der Position "von" und endend bei der Position "bis", es muß gelten: 1 <= von <= bis <= LENGTH t; entspricht: *text (t, bis - von + 1, von)*.

Beispiel für die beiden "subtext" Prozeduren:

```
TEXT CONST wort  :: "Genesender";
putline (subtext (wort, 5));
putline (subtext (wort, 6, 9))
```

Ausgabe:

```
sender
ende
```

TEXT OP SUB (TEXT CONST t, INT CONST p)

liefert aus dem Text t den 1 Zeichen langen Teiltext an der Position p; es muß gelten: 1 <= p <= LENGTH t; dieser Operator entspricht dem Prozeduraufruf *subtext (t, p, p)*.

PROC replace (TEXT VAR t, INT CONST p, TEXT CONST neu)

ersetzt einen Teiltext von t, beginnend bei der Position p, durch den TEXT "neu" bis zur Position p + (LENGTH neu) - 1; es gilt: 1 <= p <= LENGTH t: die Länge von t wird durch diese Prozedur nicht verändert, d.h. ist LENGTH neu > (LENGTH t) - p + 1, so wird der Rest von "neu" abgeschnitten.

Beispiel:

```
TEXT VAR eier legendes monster  :: "Hahn";
:
:
IF eier legendes monster = "Hahn"
THEN replace (eierlegendes monster, 2, "u")
END IF;
:
:
```

PROC change (TEXT VAR t, TEXT CONST alt, neu)

ersetzt den Teiltext "alt" in Text t an der ersten Stelle seines Auftretens durch den TEXT "neu"; tritt der Teiltext "alt" nicht in t auf, so erfolgt eine Meldung; durch diese Prozedur wird die Länge von t geändert, falls LENGTH alt <> LENGTH neu; dies kann zu unerwünschten Seiteneffekten und Fehlern führen.

Beispiel:

```
TEXT VAR wort  :: "Genesender";
change (wort, "sender", "empfänger");
put (wort)
```

Ausgabe:

```
Geneempfaenger
```

INT PROC pos (TEXT CONST t1, t2)

liefert die Position in t1, bei der das erste Auftreten des Texts t2 in t1 beginnt; ist t2 nicht in t1 enthalten oder ist t1 ein niltext, so liefert die Prozedur den Wert 0.

INT PROC pos (TEXT CONST t1, t2, INT CONST anfang)

analog der obigen Prozedur pos, jedoch beginnt hier die Suche in t1 erst bei der Position "anfang"; ist von dieser Position bis zum Ende von t1 der Text t2 nicht in t1 enthalten, ist anfang > LENGTH t1 oder ist t1 ein niltext, so liefert die Prozedur den Wert 0.

Beispiel für "change" und "pos":

```
TEXT VAR zeile;
getline (zeile);
WHILE NOT eof REP
    setze umlaute um;
    putline (zeile);
    getline (zeile)
END REP.

setze umlaute um:
   INT VAR anfang :: pos (zeile, "e", 2),
   WHILE anfang > 0 REP
      TEXT CONST vokal vermutet :: zeile SUB (anfang - 1);
      IF   vokal vermutet = "ä"
           THEN change (zeile, "ae", "ä")
      ELIF vokal vermutet = "ö"
           THEN change (zeile, "oe", "ö")
      ELIF vokal vermutet = "ü"
           THEN change (zeile, "ue", "ü")
      END IF;
      anfang := pos (zeile, "e", anfang + 1)
   END REP
```

13.2. Standard der abstrakten Datentypen

Im definierten Standard sind als abstrakter Datentyp auch die komplexen Zahlen vorgegeben, da sie in mathematischen und physikalischen Anwendungen häufig benötigt werden. Zugleich sieht man an diesem Teil der Standardpakete, wie durch die Definition des Datentyps COMPLEX und dazugehörige Operationen mittels des Paket-Konzepts eine faktische Erweiterung des Sprachumfangs von ELAN erreicht wird.

13.2.1. Datentyp COMPLEX

Der Datentyp COMPLEX setzt sich aus dem Realteil ("re") und Imaginärteil ("im") einer komplexen Zahl zusammen:

TYPE COMPLEX = STRUCT (REAL re, im)

Vorgegebene Konstanten:

```
COMPLEX CONST  complex null = COMPLEX : (0.0, 0.0),
               complex eins = COMPLEX : (1.0, 0.0),
               complex  i   = COMPLEX : (0.0, 1.0)
```

Denotierungsprozeduren:

COMPLEX PROC complex (REAL CONST r, i)

liefert einen COMPLEX-Wert mit Realteil r und Imaginärteil *i* (kartesische Koordinatendarstellung).

COMPLEX PROC complexpolar (REAL CONST abs, phi)

liefert einen COMPLEX-Wert aus der angegebenen Polarkoordinaten-Darstellung, die dann in die kartesische Koordinatendarstellung umgewandelt wird.

Teilkonvertierung:

REAL PROC realpart (COMPLEX CONST z)

liefert den Realteil von z.

REAL PROC imagpart (COMPLEX CONST z)

liefert den Imaginärteil von z.

REAL PROC abs (COMPLEX CONST z)

liefert den Betrag (Norm) von z.

REAL OP ABS (COMPLEX CONST z)

wie oben.

Beispiel:

```
COMPLEX CONST z  :: complex (1.0, 1.0);
:
put (ABS z);  (* entspricht hier: put (sqrt (2.0)) *)
:
```

Ausgabe:

```
1.414213562E0
```

:

REAL PROC phi (COMPLEX CONST z)

liefert den Winkel (in Bogenmaß) der Polarkoordinatendarstellung von z.

Beispiel:

```
COMPLEX CONST z   0  :: complex eins,
              z  90  :: complex i,
              z 180  :: complex (-1.0, 0.0),
              z 270  :: complex (0.0, -1.0);
:
put (phi (z0));      (* phi (z0)    = 0.0 *)
put (phi (z90));     (* phi (z90)   = pi/2.0 *)
put (phi (z180));    (* phi (z180)  = pi  *)
put (phi (z270));    (* phi (z270)  = 1.5 * pi *)
:
```

Ausgabe:

```
:
 0.0   1.570796327E0   3.141592654E0   4.712388980E0
:
```

REAL PROC phid (COMPLEX CONST z) ;)

liefert den Winkel wie zuvor, jedoch in Winkelgrad ("degree").

Vergleichsoperatoren:

BOOL OP = (COMPLEX CONST z1, z2)

testet z1 und z2 auf Gleichheit und liefert TRUE, falls z1 und z2 in den Realteilen und in den Imaginärteilen übereinstimmen, andernfalls FALSE (entspricht der Bedingung: (re.z1 = re.z2) AND (im.z1 = im.z2)).

BOOL OP <> (COMPLEX CONST z1, z2)

testet auf Ungleichheit und entspricht der Negation des vorigen Operators.

Da auf den komplexen Zahlen keine Ordnung vorhanden ist, gibt es auch keine Operatoren <, <=, >, >= zwischen COMPLEX-Objekten.

Arithmetik:

Die Operatoren +, -, *, / für die vier Grundrechenarten sind auch für COMPLEX-Objekte so definiert, wie man ihre Wirkungsweise aus der Mathematik kennt. Ferner gibt es noch:

COMPLEX OP CONJ (COMPLEX CONST z)

liefert die zu z konjugiert komplexe Zahl
(entspricht: COMPLEX: (re.z, -im.z)).

Ein-/Ausgabe:

PROC get (COMPLEX VAR z)

liest in z einen COMPLEX-Wert von der Standardeingabe "sysin" (vergl. auch 12.2.).

PROC put (COMPLEX CONST z)

gibt den COMPLEX-Wert von z auf die Standardausgabe "sysout" aus.

Die Darstellung eines COMPLEX-Wertes für die Eingabe und die Ausgabe sind implementationsabhängig; häufig wird ein COMPLEX-Wert durch zwei REAL-Darstellungen, durch blank getrennt, jedoch nicht weiter gekennzeichnet, repräsentiert.

Beispiel zum Standard für COMPLEX:

```
PACKET  widerstandsberechnung DEFINES
        WIDERSTAND, REIHE, PARALLEL, get, put:

TYPE WIDERSTAND = COMPLEX.
(* Fuer einige Anwendungen braucht die Feinstruktur nur aus REAL bestehen *)

WIDERSTAND OP REIHE (WIDERSTAND CONST r1, r2):
     WIDERSTAND: (CONCR (r1) + CONCR (r2))
END OP REIHE;

WIDERSTAND OP PARALLEL (WIDERSTAND CONST r1, r2):
     WIDERSTAND: (CONCR (r1) * CONCR (r2) / (CONCR (r1) + CONCR (r2)))
END OP PARALLEL;

PROC put (WIDERSTAND CONST (r))
     put (CONCR (r))
END PROC put;

PROC get (WIDERSTAND VAR r):
     COMPLEX VAR z;
     get (z);
     r := WIDERSTAND: (z)
END PROC get
END PACKET widerstandsberechnung;

(* Anwenderprogramm:* )
   ROW 7 WIDERSTAND VAR r;
   widerstaende einlesen;
   gesamtwiderstand berechnen;
   gesamtwiderstand ausgeben.

   widerstaende einlesen:
   INT VAR i;
         FOR i FROM 1 UPTO 7 REP
         get (r [i] )
         END REP.
```

```
gesamtwiderstand berechnen:
WIDERSTAND VAR rgesamt;
   rgesamt := (r [1]  PARALLEL r [2]  ) REIHE
              r [3]  REIHE
             (r [4]  PARALLEL r [5] PARALLEL r [6] PARALLEL r [7] ).

gesamtwiderstand ausgeben:
   put ("Der Gesamtwiderstand der Schaltung betraegt:");
   put (rgesamt)
```

13.3. Standard der mathematischen Routinen und Konstanten

Gerade bei den mathematischen Routinen existieren sehr viele, die zwar nicht zum definierten Standard gehören, jedoch eine sehr nützliche Erweiterung darstellen. Außerdem sei noch erwähnt, daß im erweiterten Standard die trigonometrischen Funktionen, die Exponential- und Logarithmusfunktionen sowie die Quadratwurzel auch für INT-Parameter deklariert sind, ohne daß wird diese Prozedurköpfe hier noch zusätzlich aufführen.

Konstanten:

```
REAL CONST  pi ::  3.1415926535897932846,
            e  ::  2.7182818284590452354.
```

Die Genauigkeit ist implementationsabhängig.

Quadratwurzel:

```
REAL PROC sqrt (REAL CONST x)
COMPLEX PROC sqrt (COMPLEX CONST z)  *)
```

Trigonometrische Funktionen:

REAL PROC sin (REAL CONST x)

Sinus, Parameter im Bogenmaß

REAL PROC sind (REAL CONST x) *)

Sinus, Parameter in Winkelgrad

REAL PROC cos (REAL CONST x)

Kosinus, Parameter im Bogenmaß

REAL PROC cosd (REAL CONST x) *)

Kosinus, Parameter in Winkelgrad

REAL PROC tan (REAL CONST x)

Tangens, Parameter im Bogenmaß

REAL PROC tand (REAL CONST x) *)

Tangens, Parameter in Winkelgrad

REAL PROC arctan (REAL CONST x)

Arcustangens, Ergebnis im Bogenmaß

REAL PROC arctand (REAL CONST x) *)

Arcustangens, Ergebnis in Winkelgrad

Exponential- und Logarithmusfunktionen:

REAL PROC exp (REAL CONST x)

Eulerfunktion: e hoch x.

REAL PROC ln (REAL CONST x)

natürlicher Logarithmus (Basis: e)

REAL PROG log2 (REAL CONST x) *)

binärer Logarithmus (Basis: 2)

REAL PROC log10 (REAL CONST x) *)

dekadischer Logarithmus (Basis: 10)

Zufallszahlengenerator:

REAL PROC random

liefert gleichverteilte Zufallszahlen zwischen 0.0 und 1.0

INT PROC random (INT CONST min, max)

liefert gleichverteilte Zufallszahlen vom Typ INT; der Wertebereich liegt zwischen "min" <= "max", einschließlich der Grenzen.

REAL PROC random (REAL CONST min, max) *)

wie zuvor, jedoch für Zufallszahlen vom Typ REAL.

REAL PROC norm random (REAL CONST mean, sdev) *)

liefert vormalverteilte Zufallszahlen mit dem Mittelwert "mean" und der Standardabweichung "sdev".

PROC initialize random (REAL CONST x)

initialisiert den Zufallszahlengenerator mit 0 <= x <= 1 (anderenfalls keine Initialisierung); hierdurch werden Folgen von Zufallszahlen reproduzierbar gemacht.

Beispiel (einem Würfelspiel):

```
INT VAR wuerfelnummer, ...;
ROW 5 INT VAR augenzahl;
ROW 5 BOOL VAR wuerfel frei;
:
:
wuerfeln;
:
:
wuerfeln:
FOR wuerfelnummer FROM 1 UPTO 5
REP
    IF wuerfel frei [wuerfelnummer]
    THEN augenzahl [wuerfelnummer] := random (1,6)
    END IF
END REP
:
:
```

Durch das Refinement "wuerfeln" wird ein echtes Würfeln mit den noch im Spiel befindlichen Würfeln simuliert. Da es bei diesem Spiel nicht sinnvoll ist (sondern eher Langeweile bewirken würde), jedesmal bei Spielbeginn dieselbe Würfelfolge zu erhalten, wird der Zufallszahlengenerator über die Tageszeit mittels einer speziellen Prozedur "time of day" (implementationsspezifisch, 5. und 6. Stelle sind die Sekunden) initialisiert:

```
:
initialize random (0.01 * real (int (subtext (time of day, 5, 6))));
:
```

13.4. Standard der Konvertierungen

Konvertierungsprozeduren wandeln den Wert eines Datenobjekts in eine Textrepräsentation um oder liefern umgekehrt aus einem TEXT-Objekt einen Wert eines gewünschten Typs.

13.4.1. Umwandlung in einen Text

Diese Prozeduren liefern eine Textrepräsentation der als Parameter verwendeten Datenobjekte im Standardformat mit möglichst kurzer Länge. Nicht negative Zahlen erhalten ein führendes blank.

```
TEXT PROC text  (INT  CONST x)
TEXT PROC text  (REAL CONST x)
TEXT PROC text  (BOOL CONST w)  ;)
TEXT PROC text (COMPLEX CONST z)
```

Beispiel:

```
TYPE TRIPEL =STRUCT (REAL x, y, z);
:
PROC put (TRIPEL CONST v):
TEXT CONST ausgabetext  :: "(" + text (v.x)
     + "|" + text (v.y)
     + "|" + text (v.z) + ")";
put (ausgabetext)
END PROC put;
:
```

Um die TEXT-Objekte "(", "|" und ")" mit den einzelnen REAL-Objekten der Feinstruktur des Datenobjekts "v" konkatenieren zu können, müssen diese REAL-Objekte erst in TEXT-Objekte umgewandelt werden, da der Konkatenationsoperator "+" nur auf zwei TEXT-Operanden erklärt ist. Zwar könnte man scheinbar dieses Problem der Konkatenation umgehen, wenn man den "ausgabetext" nicht erst aufbaut, sondern gleich seine "Einzelteile" in sieben verschiedenen put-Anweisungen ausgibt. Doch führt dieses zum Auseinanderreißen der TRIPEL-Repräsentation, falls ein Teil der Ausgabe so in die nächste Zeile gerät. Im vorherigen Beispiel ist dies nicht möglich.

Weitere Umwandlungsprozeduren liefern einen TEXT fest vorgegebener Länge, rechtsbündig eingetragen:

TEXT PROC text (INT CONST x, INT CONST laenge)
TEXT PROC text (REAL CONST x, INT CONST laenge)
TEXT PROC text (REAL CONST x, INT CONST laenge, nachkomma)
Enthält ein TEXT mehr Zeichen als die "laenge" angibt, so ist das Ergebnis vom Standard her nicht definiert. In Anlehnung an Implementationen anderer Programmiersprachen liefert der erweiterte Standard bei dieser Längenüberschreitung einen Text der angegebenen Länge, der nur aus "*" besteht.

Beispiel:

```
ROW 5 ROW 5 REAL VAR wertetafel;
INT VAR zeile, spalte;
:
:
FOR zeile FROM 1 UPTO 5 REP
    gib zeile aus
END REP.

gib zeile aus:
    FOR spalte FROM 1 UPTO 5 REP
        put (text (wertetafel [zeile] [spalte], 8, 2))
    END REP;
    line
```

Hierdurch erreicht man, daß in der Ausgabe die Zahlen rechtsbündig untereinander stehen; zur Veranschaulichung folgende fiktive Ausgabe:

```
    1.00 -1234.56     0.01    -4.70    98.45
-9876.54     1.11 -1234.56 99999.91 -2468.01
  111.25 -1593.77    77.77   367.93   138.77
  -37.03   593.77  -158.33 -8743.00 -2468.02
-3703.21 59377.14 -1583.29   -87.43     0.00
```

13.4.2. Erkennen numerischer Werte

Die Prozeduren zum Erkennen numerischer Werte sollen eine Repräsentation eines bestimmten Datentyps erkennen und in einen Wert dieses Typs umwandeln. Bei einer unzulässigen Repräsentation (falsche Zeichen, falsche Abfolge von Zeichen oder unzulässige Wertgröße) findet keine Umwandlung statt; durch die Prozeduren des erweiterten Standards wird dann der Programmlauf mit Fehlermeldung beendet; in diesen Implementationen fehlt folglich auch die am Schluß dieses Abschnitts aufgeführte Prozedur zur Abfrage auf Konvertierungsfehler.

Die folgenden Prozeduren beginnen die Erkennung direkt an der ersten Stelle im TEXT:

```
INT     PROC int     (TEXT CONST t)
REAL    PROC real    (TEXT CONST t)
BOOL    PROC bool    (TEXT CONST t)  *)
COMPLEX PROC complex (TEXT CONST t)
```

Beispiel (Rückführung der Prozedur "get" für INT-Objekte auf "get" für TEXT-Objekte):

```
PROC  get  (INT VAR x):
TEXT  VAR t;
get   (t);
x  := int (t)
END PROC get
```

Außerdem existieren Prozeduren, die die Erkennung bei einer Stelle p im Text beginnen:

```
INT     PROC  int     (TEXT CONST t, INT CONST p)
REAL    PROC  real    (TEXT CONST t, INT CONST p)
BOOL    PROC  bool    (TEXT CONST t, INT CONST p)  *)
COMPLEX PROC  complex(TEXT CONST t, INT CONST p)
```

Ein Konvertierungsfehler, wie er auch zu Anfang des Abschnitts erläutert wurde, kann durch folgende Prozeduren abgefragt werden:

```
BOOL PROC  last conversion ok
```

13.5. Besonderheiten

Für gewisse ELAN-Implementationen existieren als Besonderheiten noch gewisse Operationen, die zwar dort in den erweiterten Standard aufgenommen worden sind, jedoch sehr stark von der Rechnerkonfiguration abhängen. So gibt es beispielsweise Zeit-Operationen, die das aktuelle Tagesdatum und Zeit liefern, eine interne Uhr ein- und ausschalten und deren Zeitdifferenz liefern. Außerdem wird die Möglichkeit gegeben, durch spezielle Positionierungsoperationen die Ausgabe auf Sichtgerät-Bildschirmen über die üblichen Ausgabearten hinaus zu steuern. Da diese Operationen aber in hohem Maße installationsabhängig und daher auch meist nicht in ELAN geschrieben sind, können sie nur sehr bedingt von einer ELAN-Implementation auf die andere übertragen werden.

Soweit es jedoch möglich ist, möchten wir anraten, Erweiterungen des Standards - zumindest als zusätzliche Version - in ELAN zu schreiben, um diese Erweiterung auch auf andere Implementationen zu übertragen und möglichst vielen Benutzern zugänglich zu machen.

Aus dem Gesagten sieht man, daß unter Umständen bei verschiedenen Implementationen auch verschieden mächtige Standard-Pakete auftreten können, die zum Teil bei weitem die hier angeführten Operationen überschreiten (sogenannte Supersets). Andererseits ist manchmal nicht der gesamte definierte Standard, der ja sowieso nur eine Empfehlung und kein unbedingtes "Muß" darstellt, sondern nur ein "Subset" vorhanden. Auf jeden Fall raten wir dem Benutzer, sich eingehend über die Standard-Pakete in der von ihm benutzten ELAN-Implementation zu informieren. Zu diesem Zweck sollte jede Installation, für die ein ELAN-Compiler implementiert wurde, ein Listing der Standardpakete sowie eine Kurzfassung der verfügbaren Prozeduren, Operatoren, Konstanten und Datentypen bereithalten.

Das beides u. U. notwendig ist, wollen wir am folgenden Beispiel zeigen. Der Operator MOD für REAL-Werte kann durch

```
REAL OP MOD (REAL CONST a,b):
     real (int(a) MOD int (b))
END OP MOD
```

oder aber auch durch

```
REAL OP MOD (REAL CONST a,b):
     (a/b - trunc (a/b)) * b
END OP MOD
```

realisiert werden, sofern der definierte Standard nicht eine Realisierung explizit vorschreibt.

Abschließend seien noch zwei Prozeduren erwähnt, die zwar auch zum definierten Standard gehören, jedoch im "normalen" Programm nicht verwendet werden sollten, da sie sich fast immer durch das LEAVE-Konstrukt umgehen lassen, während sie in "Produktionsprogrammen" und Erweiterungen der Standardpakete eine sinnvolle Anwendung finden können.

```
PROC  stop
PROC  errorstop
```

Beide Prozeduren beenden den Programmlauf, wobei die zweite Prozedur noch zusätzlich versucht, eine "post-mortem"-Behandlung durchzuführen, d.h. eine Behandlung des Programmabbruchs, wie sie bei Laufzeitfehlern (z.B. Arithmetik-Überlauf) auftritt, die dem Benutzer gewisse Rückschlüsse auf den Grund des Abbruchs und die Unit, in der der Abbruch geschah, sowie eventuell eine Rückverfolgung bei Schachtelungen erlaubt. Bei manchen ELAN-Implementationen kann die Prozedur "errorstop" noch eine zusätzliche Meldung ausgeben und hat dort den folgenden Prozedurkopf:

```
PROC  errorstop (TEXT CONST meldung)  )
```

Beispiel (für "errorstop" und "maxtextlength"):

```
TEXT PROC  text (TEXT CONST string,
                 INT CONST laenge,
                 BOOL CONST abbruch erlaubt):

   IF    laenge > LENGTH  string
   THEN  auffuellen mit blanks
   ELIF  laenge < LENGTH  string
   THEN  abschneiden
   ELSE  string
END IF.

auffuellen mit blanks:
   IF    laenge <= maxtextlength
   THEN  string + (laenge - LENGTH string)* " "
   ELSE  eventuell abbruch;
         string + (maxtextlength - LENGTH string)* " "
END IF.

eventuell abbruch:
   IF    abbruch erlaubt
   THEN  putline ("Angegebene Laenge > maxtextlength in text-Proc");
         errorstop
END IF.

abschneiden:
   subtext (string, 1, laenge)
END PROC text
```

Anhang A: Schlüsselwörter mit einer festen Bedeutung ("bolds") (mit Angabe alternativer Repräsentationen)

CASE
CONCR
CONST
DEFINES
DOWNTO
ELIF
ELSE
ENDIF, END IF, FI
ENDOP, END OP, ENDOPERATOR, END OPERATOR
ENDPACKET, END PACKET
ENDPROC, END PROC, ENDPROCEDURE, END PROCEDURE
ENDREP, END REP, ENDREPEAT, END REPEAT, PER
ENDSELECT, END SELECT
FOR
FROM
IF
LEAVE
LET
OF
OP, OPERATOR
OTHERWISE
PACKET
PROC, PROCEDURE
REP, REPEAT
ROW
SELECT
STRUCT
THEN
UNTIL
UPTO
VAR
WHILE
WITH

Anhang B: Kleines ELAN-Wörterbuch

Abkürzungsvereinbarung ("shorthand declaration", LET-Konstrukt)
ermöglicht die Einführung eines Bezeichners für Denoter ↑einfacher Datentypen sowie für Datentypen.

abstrakter Datentyp
nennt man einen ↑neudefinierten Datentyp, wenn dieser neudefinierte Datentyp sich in einem Paket befindet (also der direkte Zugriff auf seine Feinstruktur von außerhalb verboten ist) und Operationen (Operatoren sowie Prozeduren) auf den Datentyp definiert sind.

Abstraktionesebene
Eine Zusammenfassung von ↑Operationen und u. U. Datentypen, die ein Anwender zum Lösen eines Problems verwenden kann. Diese werden bei dem Prozeß der Problemlösung als elementar angesehen. In ELAN kann eine Abstraktionsebene durch ein oder mehrere ↑Pakete gebildet werden, in denen Datentypen und dazugehörige Operationen definiert werden.

Accessrecht
(auch Accessattribut genannt) ist eine Eigenschaft von ↑externen Objekten und dient zur Überprüfung des Zugriffs mit Hilfe von ↑Bezeichnern auf Datenobjekte. Dabei bedeutet das Accessrecht VAR Lese-/Schreibrecht, während CONST nur Leserecht beinhaltet.

Ausdrucks-Sprache
In einer Ausdrucks-Sprache ("expression language") liefert jedes Sprachelement einen Wert. Ein ausgezeichneter Wert, der oft als "void" (ALGOL68) bezeichnet wird, bedeutet: "liefert keinen Wert". ↑Konstrukte, die "void" liefern, entsprechen Anweisungen in anweisungsorientierten Programmiersprachen ("statement languages").

Balancierung ("balancing")
Anpassung von unterschiedlichen Accessrechten bei einer zweiseitigen Abfrage, Auswahl und Terminator (nur Anpassung von VAR nach CONST).

Bezeichner
Ein Bezeichner ist ein ↑externes Objekt, das ein ↑Objekt mit Hilfe eines Namens kennzeichnet, so daß es in einem bestimmten Kontext identifiziert werden kann.

bottom up - Methode
Bereitstellen einer (oder mehrerer übereinander gelagerter) Abstraktionsebenen in Form von Paketen oder Prozeduren, mit denen eine Lösung konstruiert wird.

Compiler (ELAN)
↑Übersetzer

consting
Anpassung von VAR nach CONST.

Datei ("file")
Sammlung von gleichartigen Datensätzen, meist auf externen Speichermedien.

Datenobjekt
Ein Datenobjekt hat in ELAN einen Typ und einen Wert.

Datentyp
Eigenschaft eines Datenobjekts in ELAN. Der Datentyp bestimmt, welche Operationen für ein Datenobjekt dieses Typs zur Verfügung stehen.

Deklaration
Vereinbarung von ↑Objekten. Jedes Objekt in ELAN muß deklariert werden.

Denoter
Wertere präsentation in einem ELAN Programmtext. Für die einfachen Datentypen (TEXT, INT, REAL, BOOL) ist die Form von Denotern vorgegeben. Für ↑abstrakte Datentypen müssen ↑Denotierungs-Prozeduren geschrieben werden.

Denotierungs - Prozedur
Prozedur, die einen (abstrakten) Datentyp liefert und deshalb an Stelle eines ↑Denoters verwendet werden kann.

direkte Datei ("direct access file")
Datei, in der Datensätze mit Hilfe eines (Satz-)Schlüssels abgespeichert werden. Mit Hilfe dieses Satzschlüssels (oft ein Integer-Wert, der eine Zeilennummer angibt) kann direkt (ohne vorangehende Information zu lesen) auf Daten zugegriffen werden, unabhängig davon, wo sich die Sätze in der Datei befinden.

display
Denoter für Reihungen

dyadischer Operator
Operator mit zwei Operanden

einfache Datentypen
sind in ELAN die Datentypen INT, REAL, TEXT und BOOL, die vorgegeben sind. Einfache Datentypen werden auch primitive, elementare oder klassisch denotierbare Datentypen (weil für sie vorgegebene Formen von ↑Denotern existieren) genannt.

Einheit
Grundbestandteil der Sprache ELAN. Entspricht einer Anweisung in anderen Programmiersprachen. Einheiten können Werte liefern. (↑Ausdrucks-Sprache)

Elaboration
Bearbeitung eines Sprachkonstrukts zur Laufzeit eines Programms.

elementare Datentypen
↑einfache Datentypen

externes Objekt
sind Zeichen und Zeichenfolgen (Worte) eines Programms. Externe Objekte werden durch einen ↑Übersetzer auf interne Objekte abgebildet. Externe Objekte können von einem Rechner nicht direkt verarbeitet werden.

Feinstruktur
Realisierung eines neudefinierten Datentyps durch andere Typen.

Generizität
Gleiche Bezeichner benennen in ELAN unter Umständen verschiedene Operatoren bzw. Prozeduren. Diese werden nach Datentyp der Operanden bzw. Parameter und deren Anzahl und Reihenfolge eindeutig identifiziert.

Grammatik
↑zweistufige Grammatik

Gültigkeitsbereich
↑range

identifier
↑Bezeichner

Interface
(Export-) Schnittstelle eines ↑Pakets. In einem Interface werden diejenigen Objekte und Datentypen aufgeführt, die nachfolgenden Programmteilen zur Verfügung stehen sollen.

interne Objekte
sind Datenobjekte und Maschinen - Anweisungen, die von einem Rechner direkt verarbeitet werden können.

klassisch denotierbare Datentypen
↑einfache Datentypen

Konkretisierer
Sprachelement, mit dem auf die ↑Feinstruktur eines neudefinierten Datentyps zugegriffen werden kann.

Konkretisierung, implizite
erfolgt durch Subskription und Selektion.

Konstruktor
Denoter für Strukturen und neudefinierte Datentypen.

Konstrukt
Sprachelement

Konstrollstrukturen
Sprachelemente, die die Abarbeitung eines Programms zur Laufzeit steuern (Abfrage, Auswahl, Schleife, Terminator).

liefern
Sprachelemente können in ELAN durch die Ausführung (↑Elaboration) ein Datenobjekt eines bestimmten Datentyps an ein anderes umfassenderes Sprachelement liefern. Man sagt auch: ein Sprachelement liefert einen Wert (von einem bestimmten Datentyp).

Modul
abgeschlossener Programmteil zur Verwendung bei der ↑Programmierung im Großen und der ↑bottom up - Methode. Ein Modul kann nur über eine Schnittstelle (↑Interface) mit anderen Programmteilen kommunizieren.

monadischer Operator
Operator mit einem Operanden

Namensüberdeckung
Bezeichner in einem Paket, der einen gleichnamigen Bezeichner aus dem extended range überdeckt. Das Objekt aus dem extended range kann in diesem Fall nicht angesprochen werden.

neudefinierter Datentyp
Durch ↑Typ-Vereinbarung wird ein neuer Datentyp durch bereits vorhandene Datentypen festgelegt. Für einen neudefinierten Datentyp sind keine Operationen (Operatoren und Prozeduren) gegeben, mit Ausnahme der ↑Zuweisung.

Objekte
sind Datenobjekte und ↑Operationen in ELAN.

Operator
Operation auf Datenobjekten. Bei der Anwendung eines Operators wird die "infix"-Notation verwendet, d.h. der Operator wird vor seinem Operanden (bei ↑monadischem Operator) bzw. zwischen die Operanden (bei ↑dyadischem Operator) geschrieben. Operatoren können (aber müssen nicht) ein Resultat liefern (Accessrecht CONST).

Operation
ist eine Prozedur oder ein (auch neudefinierter) Operator.

Paket (PACKET)
↑Modul in ELAN. Ein Paket dient in ELAN zur Zusammenfassung von ↑Operationen und Konstanten, zur Realisierung von ↑abstrakten Datentypen und Abstraktionsebenen.

primitive Datentypen
↑einfache Datentypen

Priorität von Operatoren
legt die Reihenfolge der Ausführung von Operatoren in einem Ausdruck fest.

Programmieren im Kleinen ("programming in the small")
Realisierung eines Algorithmus nach der ↑top down - Methode.

Programmieren im Großen ("programming in the large")
Definition von Bausteinen oder ↑Abstraktionsebenen durch die ↑bottom up - Methode.

Prozedur
Ein Stück Programmtext, der mit Parametern versehen werden kann. Eine Prozedur kann Resultate beliebigen Typs in ELAN liefern. In einer ELAN - Prozedur können (lokale) Datenobjekte deklariert und Refinements sowie LET - Vereinbarungen verwendet werden. Es sind nur parallele Prozeduren möglich.

range
In ELAN existieren drei Gültigkeitsbereiche: Prozedur, Paket und der über die Schnittstelle eines Pakets auf nachfolgende Programmteile ausgedehnte Bereich ("extended range"). Der Gültigkeitsbereich legt fest, wo ein Bezeichner vereinbart und angesprochen werden darf.

refinement
sprachliches Mittel zur Unterstützung der ↑Programmierung im Kleinen in ELAN, in dem Teilalgorithmen Namen gegeben werden, die später erst konkretisiert werden müssen. ↑Schrittweise Verfeinerung.

Reihung
Verbund gleichartiger Datenobjekte

Schnittstelle ("interface")
Angabe zu einem Paket, welche Objekte nachfolgenden Paketen zur Verfügung stehen sollen.

schrittweise Verfeinerung
Unterteilung und Benennung eines Problems (↑top down - Methode). Die so entstandenen Teilprobleme werden wiederum unterteilt und benannt, bis eine elementare Abstraktionsebene erreicht wird.

Selektion
Zugriff auf die Komponenten von Strukturen.

sequentielle Datei ("sequential access file")
Datei, bei der die Information ausschließlich in strikter Reihenfolge der Daten bzw. Sätze gespeichert werden kann. Auf die Daten einer solchen Datei kann nur sequentiell zugegriffen werden.

Sequenz
bedeutet die aufeinanderfolgende Ausführung von Einheiten, die durch ↑statement separator getrennt werden.

SLAN
Nach dem Konzept von Prof. Koster eine Sprachfamilie, die die Bereiche Ausbildung, Anwendungs- und Systemprogrammierung abdecken soll. ELAN ist in dieser Sprachfamilie die Sprachstufe für die Zwecke der Ausbildung.

Standard-Pakete
↑Module in ELAN, die jedem Programmierer standardmäßig zur Verfügung stehen. Es wird zwischen dem definierten und dem erweiterten Standard unterschieden (definierter Standard: Standard-Pakete, wie sie ... der Beschreibung definiert sind; erweiterter Standard: implementations- bzw. installationsabhängige zusätzliche Pakete).

statement separator
trennt ↑Einheiten voneinander. Er wird durch das ";"-Zeichen repräsentiert und bedeutet: "führe nächste Anweisung aus" (↑Sequenz).

Steuerkonstrukte
↑Konstrollstrukturen

Struktur
Verbund (ungleichartiger) Datentypen

Subskription
Zugriff auf Komponenten einer Reihung (in anderen Programmiersprachen Indizierung genannt).

tag
Darstellung eines ↑Bezeichners im Programmtext.

Terminator
Anweisung für das Verlassen von ↑Refinements, Prozeduren und Operatoren.

top down-Methode
Problemlösung von "oben nach unten", d.h. ein Problem wird in (Teil-) Lösungen zerlegt, die wiederum in (Teil-)Lösungen untergliedert werden, bis eine Stufe erreicht wird, in der nur noch Elementar-Anweisungen der Sprache verwendet wird. In ELAN wird die top down-Methode durch das ↑Refinement auf der Prozedurebene unterstützt.

Typ-Vereinbarung
Anweisung in ELAN, durch die ein ↑Bezeichner für einen neuen, von allen anderen (bereits vorhandenen) unterschiedlichen Datentyp vereinbart wird.

Übersetzer (ELAN)
ein Programm, das ein ELAN-Quellprogramm in ein Programm in einer anderen Sprache (meist Maschinensprache) überführt, welches von einem Rechner ausgeführt werden kann.

unit
↑Einheit

van Wijngaarden Grammatik
↑zweistufige Grammatik

Vereinbarung
Festlegung von ↑Bezeichnern für ↑Objekte und ↑Datentypen (↑Deklaration).

vorgegebene Datentypen
↑einfache Datentypen und die von den ↑Standard-Paketen definierten Datentypen.

Zuweisung
Operator in ELAN, der den Wert des rechten Operanden dem linken Operanden zuweist (Wertetransport). Die Zuweisung ist für ↑neudefinierte Datentypen gegeben, wenn sie für die Komponenten des neudefinierten Datentyps gegeben ist. Sie kann neudefiniert werden. In diesem Fall braucht kein Wertetransport realisiert werden.

zweistufige Grammatik
besteht aus zwei (kontextfreien) Grammatiken, von denen die erste Grammatik (die "metarules") in die zweite ("hyperrules") eingesetzt wird. Dabei entsteht eine dritte (kontextsensitive) Grammatik, die ein einzelnes Programm beschreibt.

Anhang C: ELAN - Syntaxdiagramme

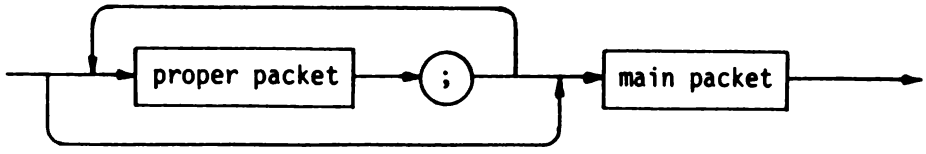
Die Syntaxdiagramme werden wie folgt gelesen:

1. Man suche den syntaktischen Begriff, über den man Informationen benötigt. Die einzelnen Syntaxdiagramme sind durch Überschriften gekennzeichnet.
2. Ein Diagramm wird am linken (oberen) Rand "betreten", darf auf jedem durch Pfeile ausgewiesenem Weg durchlaufen werden und wird rechts (unten) wieder verlassen. Dabei bedeuten:
   - jedes eingekreiste Zeichen oder in einem Kasten mit abgerundeten Ecken eingefaßte Zeichenfolge stellen Sprachelemente dar, die direkt in einem ELAN-Programm erscheinen,
   - Kästen mit rechtwinkligen Ecken stehen als Abkürzung für andere Diagramme, die man bei Bedarf nachsehen kann.
3. Einige Diagramme enthalten Schleifen, d.h. es existiert ein Weg zurück zu einem davor liegenden Teil des Diagramms. Dies wird z.B. für Wiederholung (Aufzählungen, Listen) verwendet.
4. Jedes Diagramm wird solange, der Pfeilrichtung folgend, durchlaufen, bis alle gewünschten Sprachelemente "produziert" worden sind.

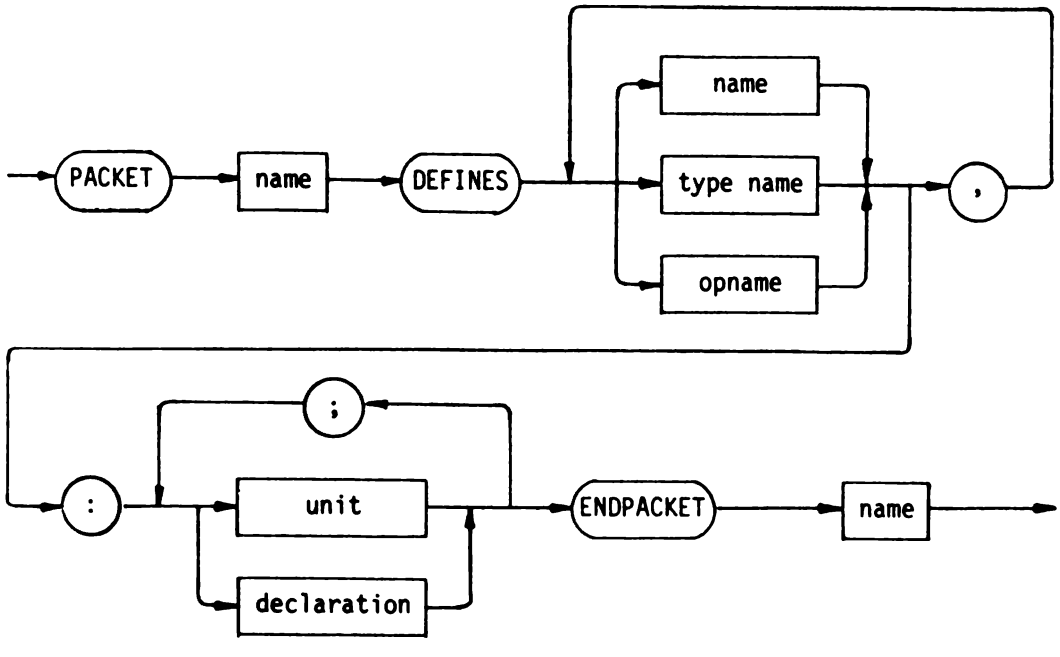
Die ELAN-Syntaxdiagramme wurden innerhalb des Forschungsprojektes Schulsprache im Fachgebiet Softwaretechnik, Institut für angewandte Informatik an der Technischen Universität Berlin erstellt.

Die Reproduktion dieser Syntaxdiagramme ist für jeden Zweck, nicht aber auszugsweise, mit Angabe der Herkunft ohne Formalitäten gestattet.

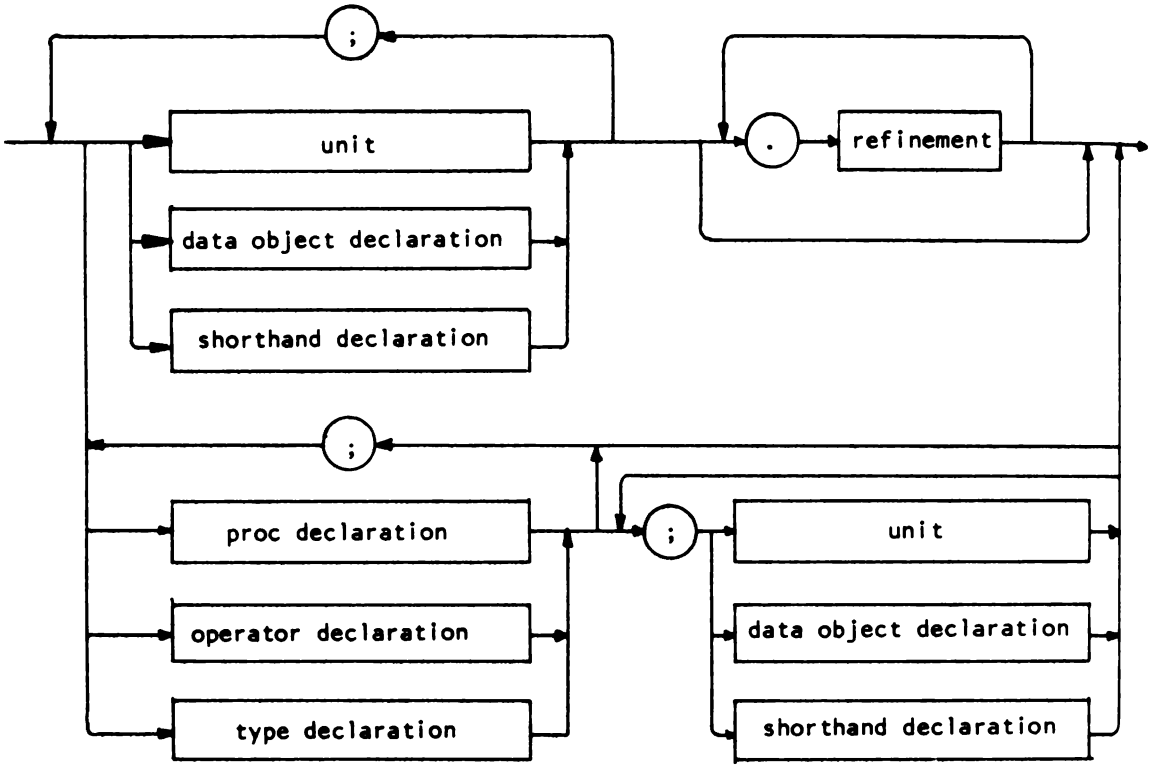
program



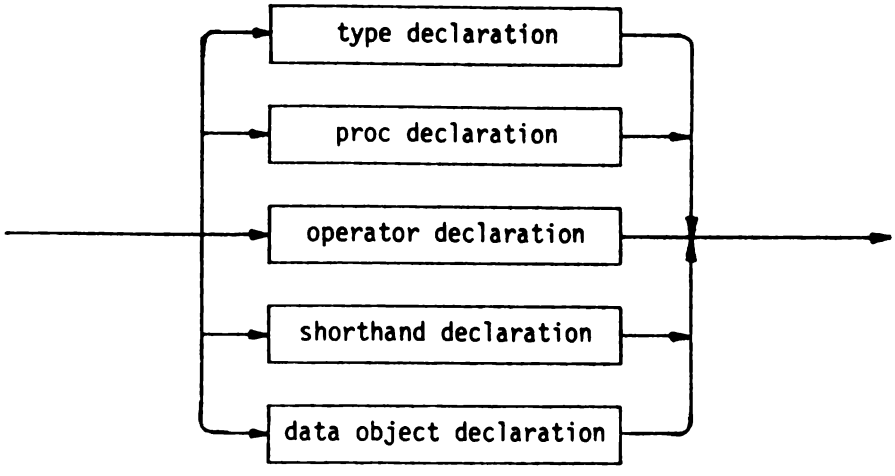
proper packet



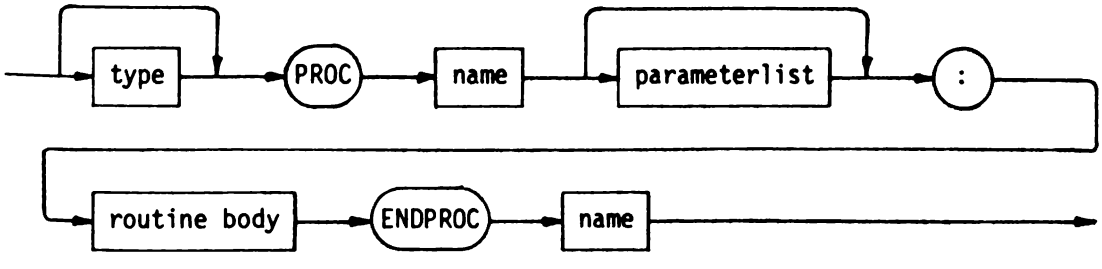
main packet


;
unit
data object declaration
shorthand declaration
.
refinement
;
proc declaration
operator declaration
type declaration
;
unit
data object declaration
shorthand declaration

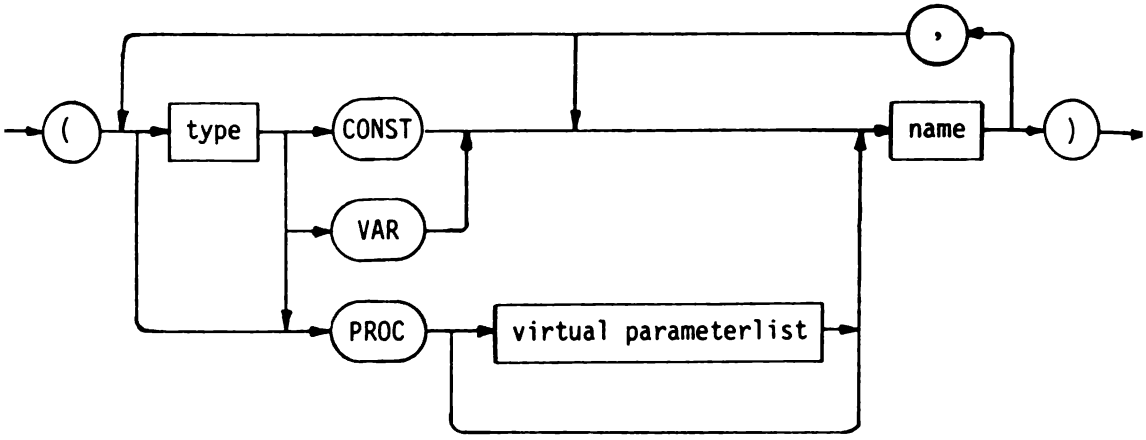
declaration



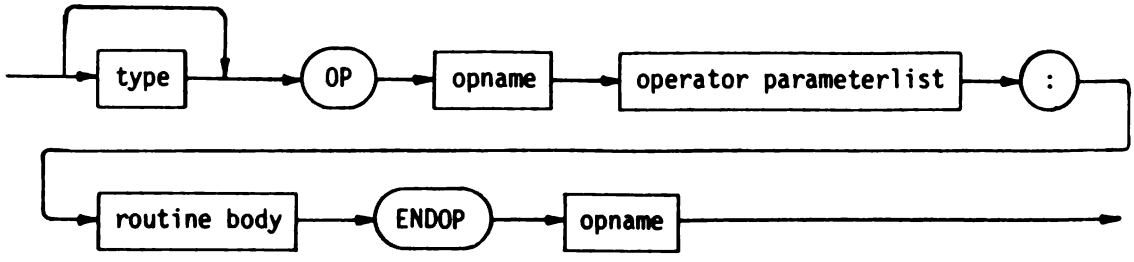
proc declaration



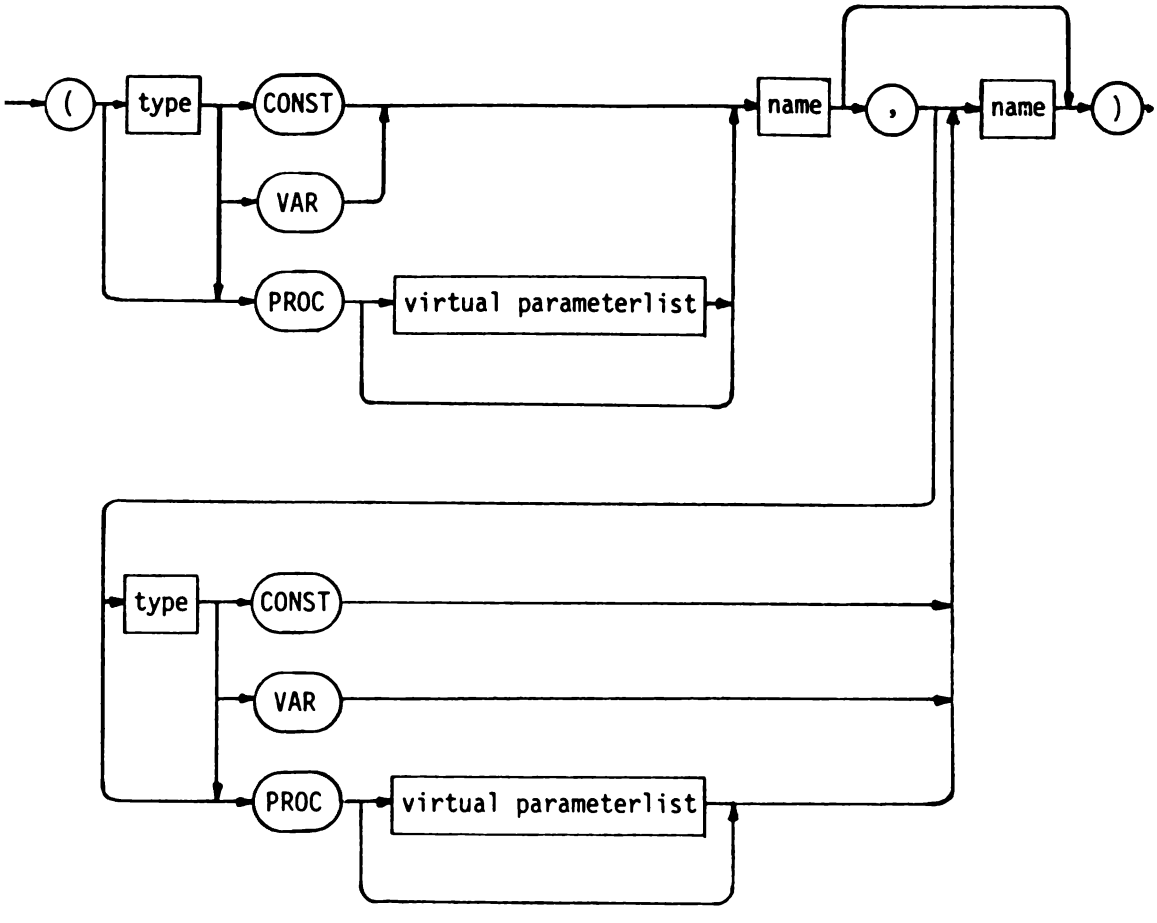
parameter list



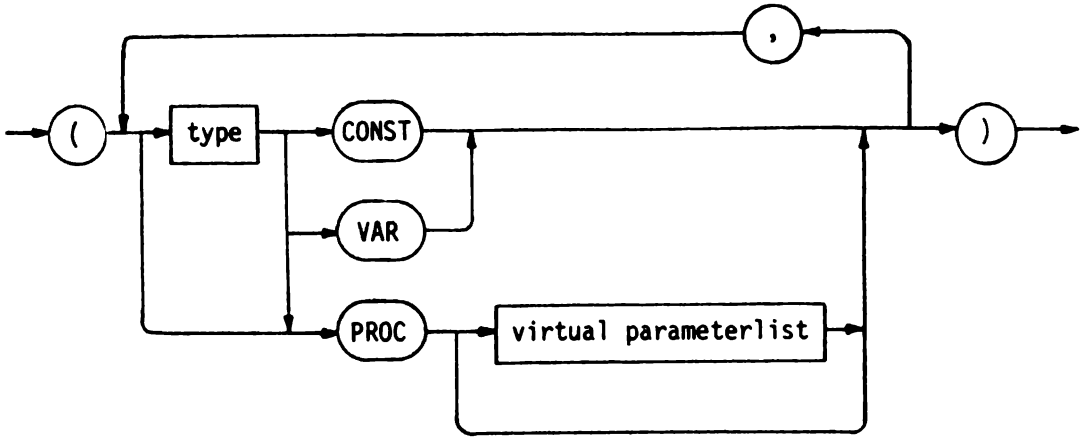
operator declaration



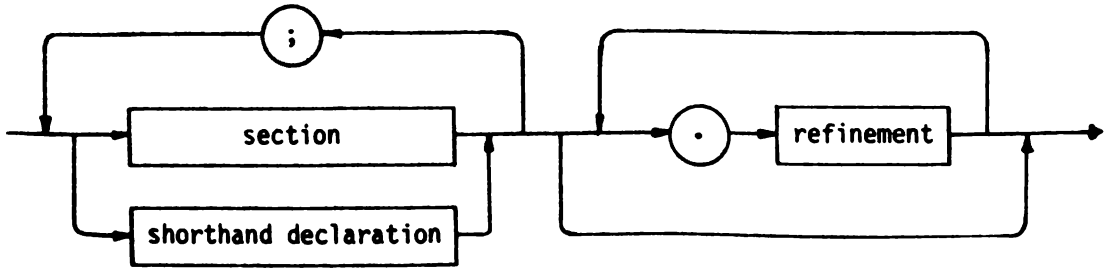
operator parameter list



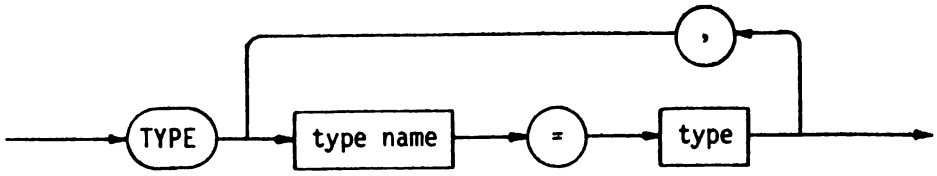
virtual parameterlist



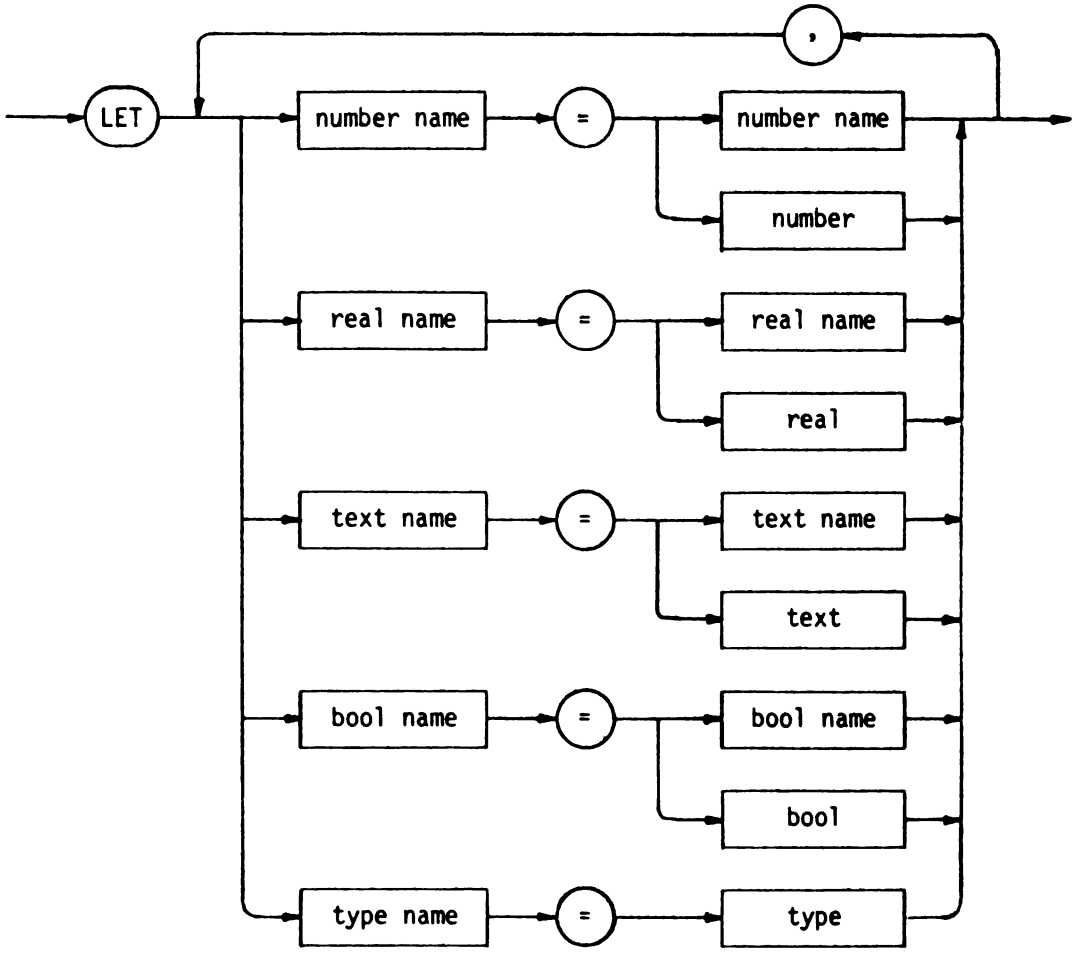
routine body



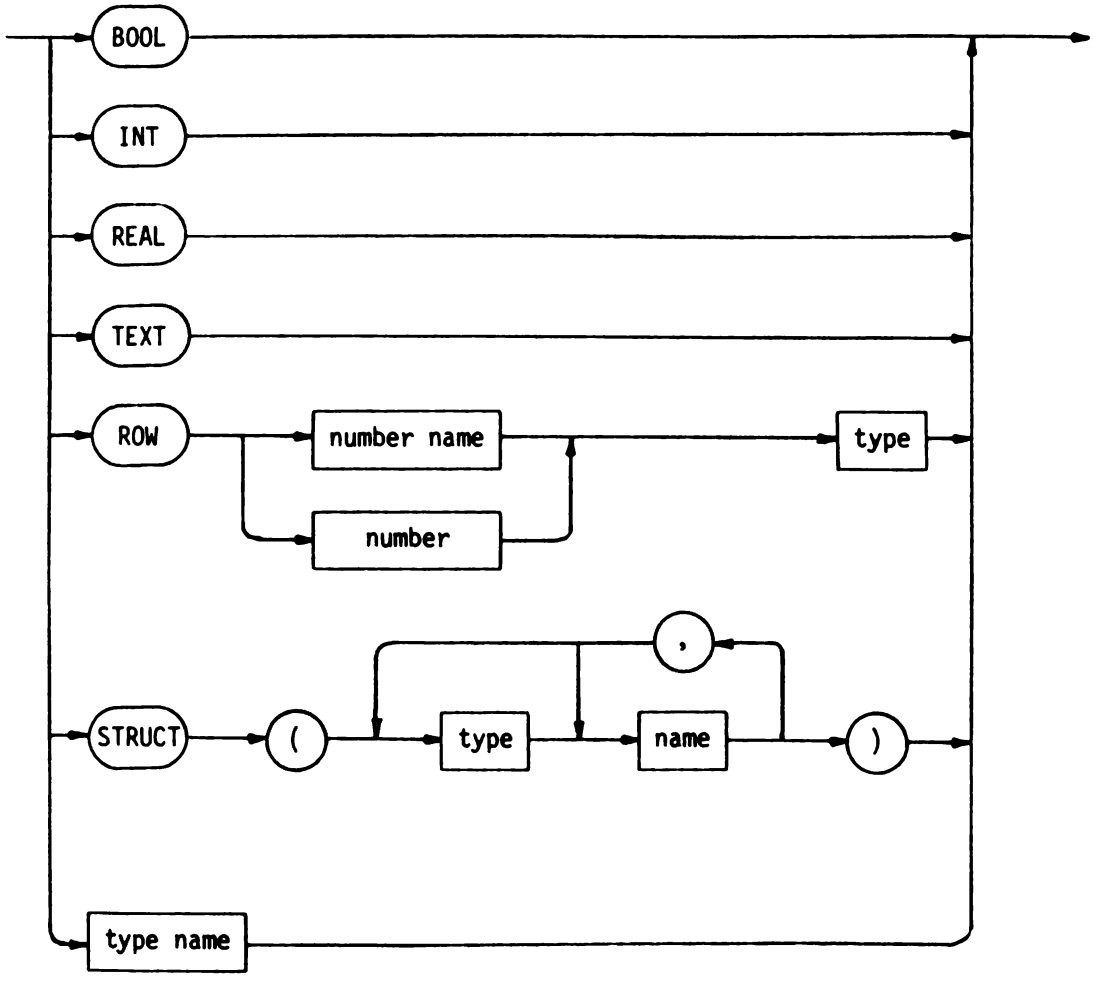
type declaration



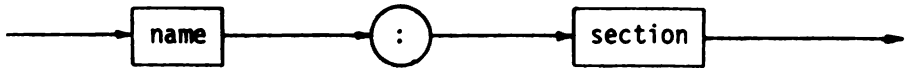
shorthand declaration



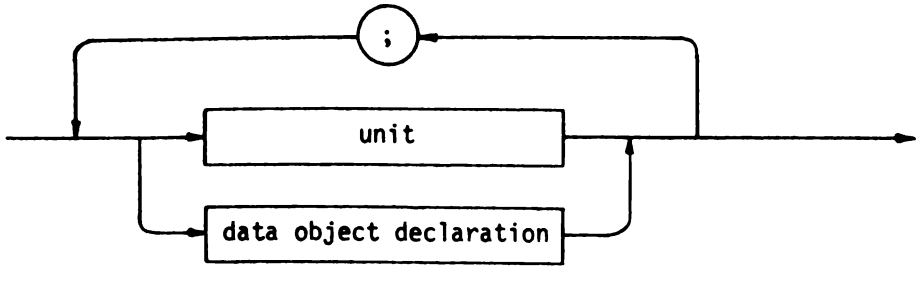
type


BOOL
INT
REAL
TEXT
ROW
number name
number
type
STRUCT
(
,
type
name
)
type name

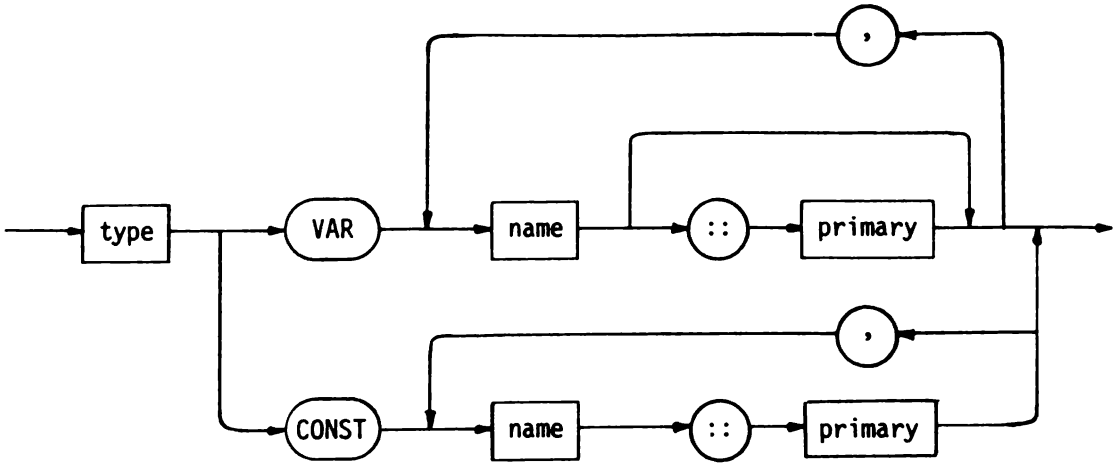
refinement



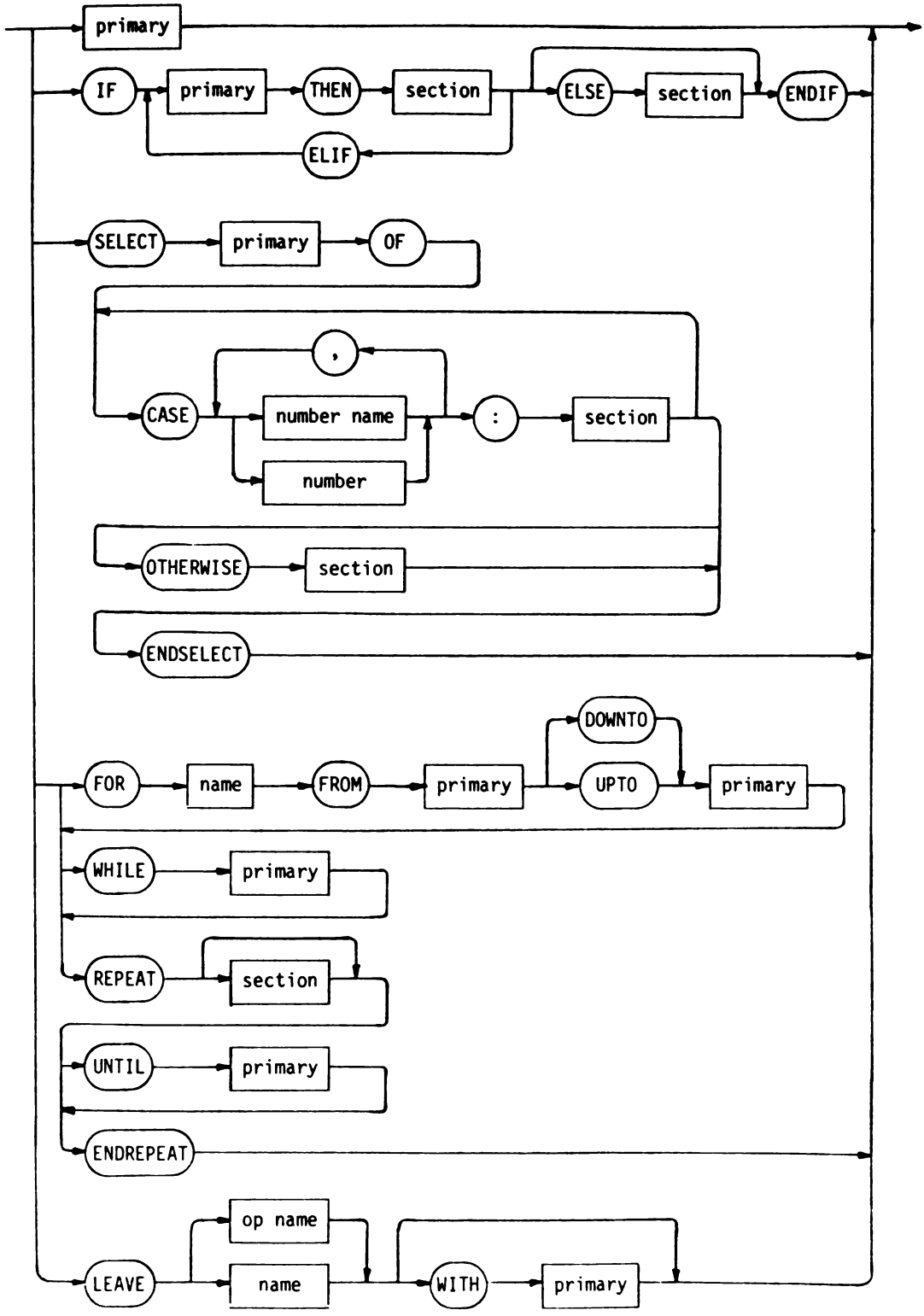
section



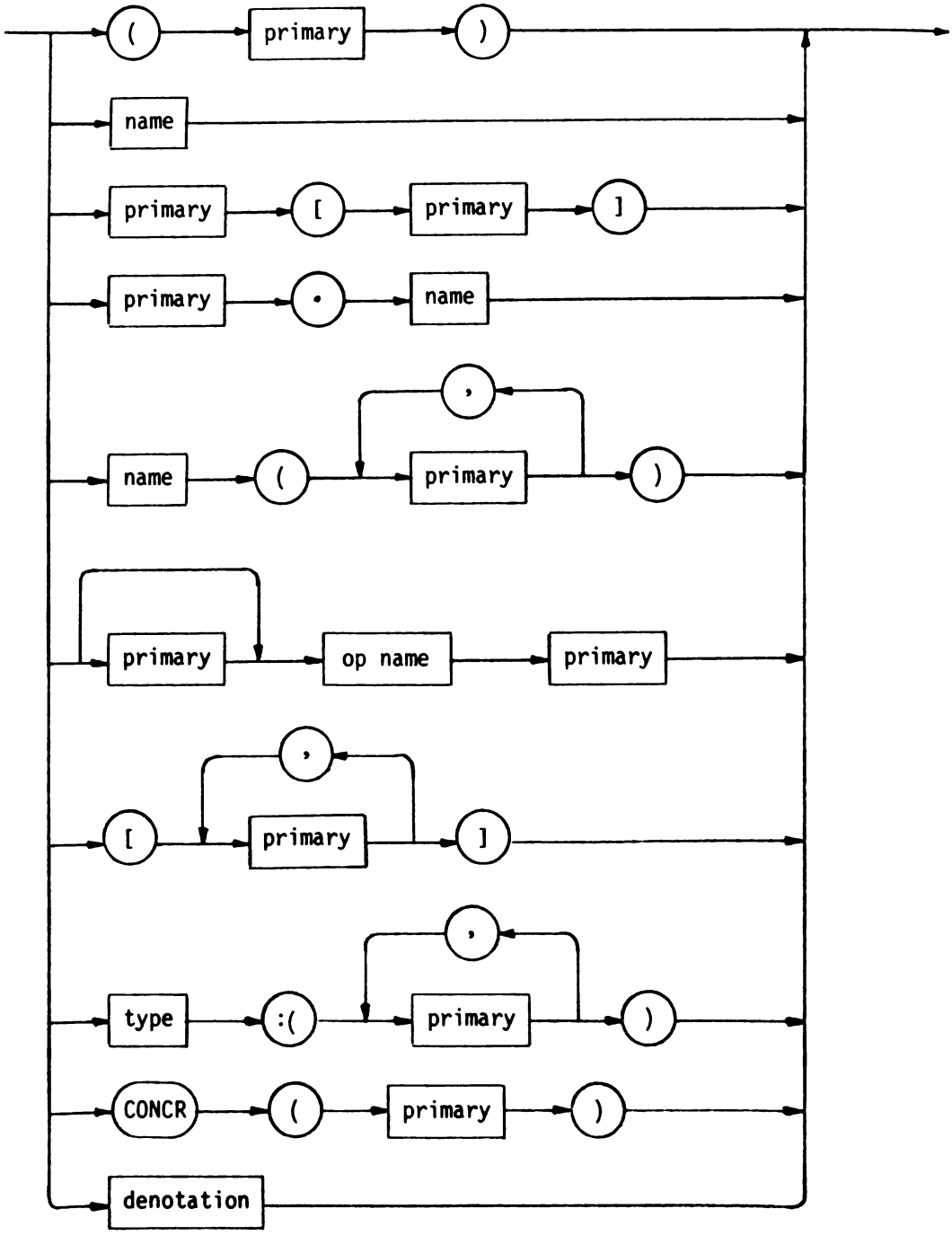
data object declaration



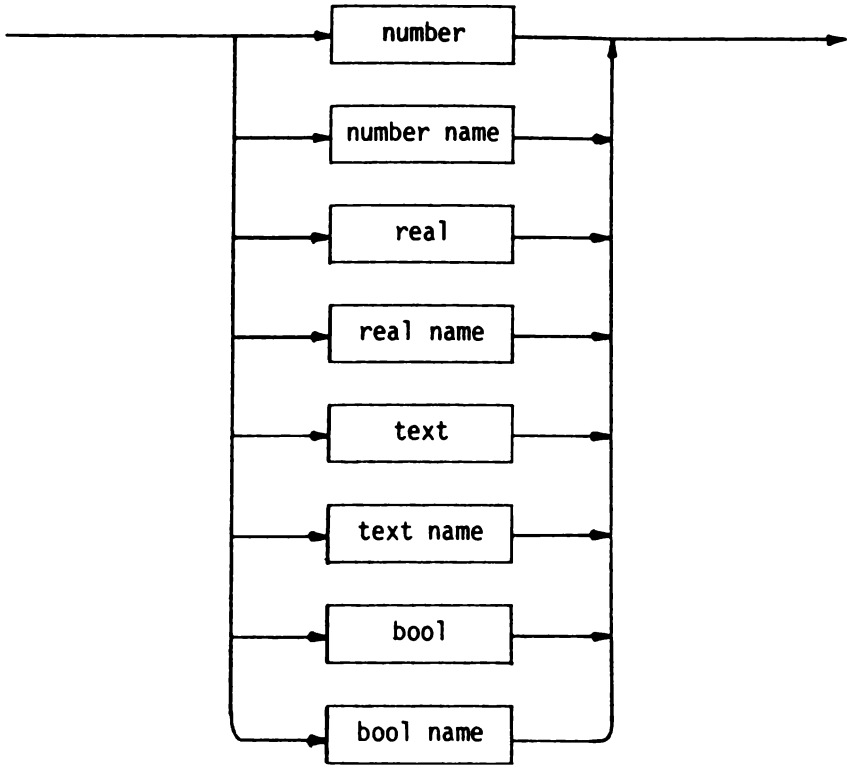
unit


primary
IF
primary
THEN
section
ELSE
section
ENDIF
ELIF
SELECT
primary
OF
CASE
,
number name
number
:
section
OTHERWISE
section
ENDSELECT
FOR
name
FROM
primary
DOWNTO
UPTO
primary
WHILE
primary
REPEAT
section
UNTIL
primary
ENDREPEAT
LEAVE
op name
name
WITH
primary

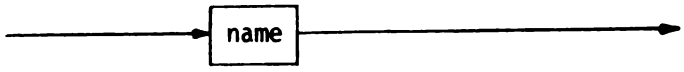
primary


(
primary
)
name
primary
[
primary
]
primary
.
name
name
(
,
primary
)
primary
op name
primary
[
,
primary
]
type
:(
,
primary
)
CONCR
(
primary
)
denotation

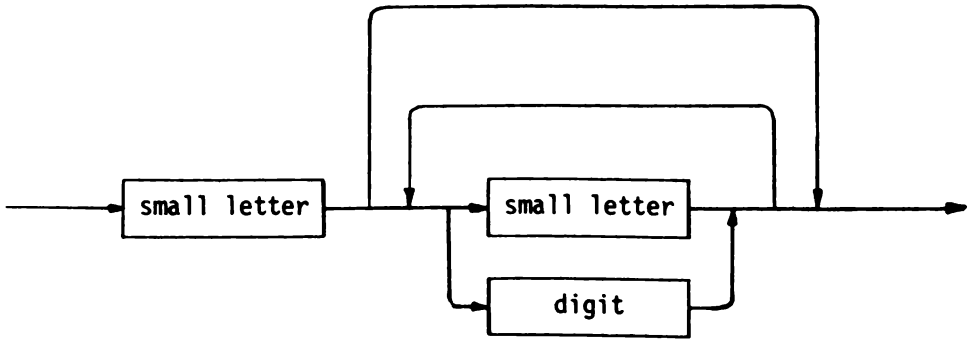
<u>denotation</u>



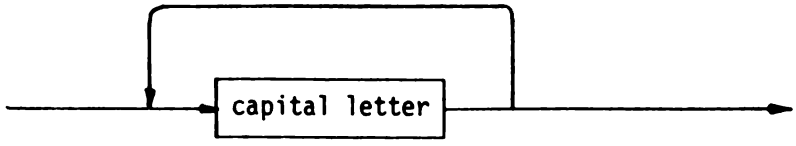
number name, real name, text name, bool name



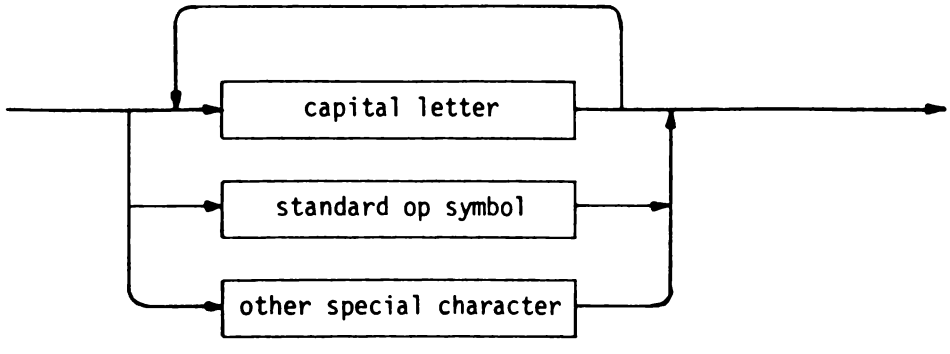
name



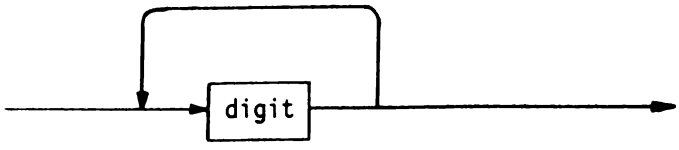
type name



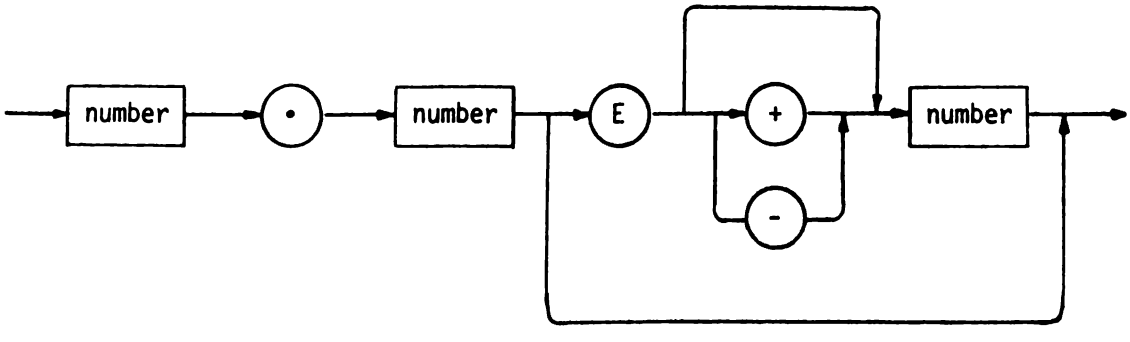
op name



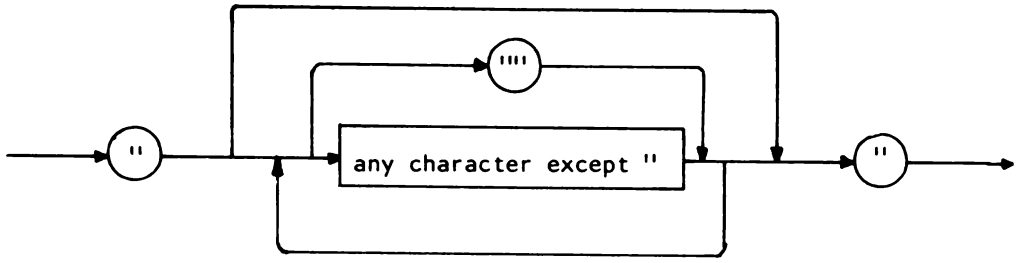
number



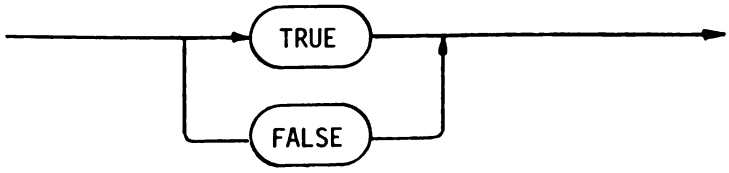
real



text



bool



Literatur

Bartling, U., Hahn, R., Liedtke, J.:
ELAN-Benutzerhandbuch. Arbeitsbericht.
HRZ der Universität Bielefeld, 1979

Bartling, U., Liedtke, J.:
Implementation des SLAN3B-Compilers. Arbeitsbericht.
HRZ der Universität Bielefeld, 1976

Bittner, M., Jäckel, J., Jähnichen, S.:
ELAN-Beispielsammlung.
Institut für angewandte Informatik.
Fachbereich 20, Technische Universität Berlin, 1979

Hahn, R., Nienaber, B.:
Probleme lösen mit dem Computer.
Teil 1: Einführung in die algorithmische Problemlösung.
Teil 2: Werkzeuge und Methoden.
Neuer Verlag, Tübingen 1978

Heyderhoff, P.:
Einführung in ELAN. Informatik-Kolleg der Gesellschaft für
Mathematik und Datenverarbeitung. Bonn, 1977

Hommel, G., Jäckel, J., Jähnichen, S., Kleine, K., Koch, W.:
Beschreibung der Programmiersprache ELAN.
Version 2.3, Bericht Nr. 78-28.
Informatik-Forschungsgruppe Softwaretechnik.
TU Berlin, 1978

Hommel, G., Jäckel, J., Jähnichen, S., Kleine, K., Koch, W., Koster, K.:
ELAN-Sprachbeschreibung. (Studientexte Informatik).
Akademische Verlagsgesellschaft, Wiesbaden 1979

Hommel, G., Jähnichen, S., Koch, W.:
SLAN - Eine erweiterbare Sprache zur Unterstützung der strukturierten
und modularen Programmierung.
Programmiersprachen 4. Fachtagung der GI.
Informatik-Fachberichte, Springer Verlag, 1976, pp. 101-110.

Hommel, G., Koster, C. H. A.:
Systematisches Programmieren mit ELAN. Interner Bericht 77-11.
Fachbereich 20, Technische Universität Berlin, 1977

Koster, C. H. A.:
Provisional CF Description of SLAN. Internes Arbeitspapier.
Fachbereich 20, Technische Universität Berlin, 1974

Liedtke, J.:
EUMEL - Ein ELAN-System für Mikroprozessoren.
ELAN - Newsletter Nr. 7.
Herausgeber: HRZ der Universität Bielefeld, 1978

STICHWORTVERZEICHNIS

Abfrage 7, 40ff.
    einseitige 40ff.
    geschachtelte 41
    -kette 7
    zweiseitige 41ff.
Abkürzungsvereinbarung 9, 48ff.
ABS 24, 104ff., 106ff.
Abstraktion 13
    schwache algorithmische 50ff.
    -ebene 10, 83ff.
    starke algorithmische 65ff.
Accessattribut 6
Accessrecht 19ff.
    Anpassung 19
    Balancierung 30ff.
    eines Resultats 23ff.
    von Denotern 18
    von Konstruktoren 35ff.
    von Reihungen 32
    von row displays 33
    von Strukturen 35ff.
    -Wandlung 29ff.
Addition 25
AND 25
Anpassung 29
Anweisungen 6
Ausdruck 26ff.
    Auswertung 26ff.
    Klammerung 27ff.
Auswahl 7, 43ff.
Auswertung eines Ausdrucks 26ff.

Balancierung 29ff.
Baumdiagramm 53
Bedingung 40
Betrag 24
Bezeichner 14ff.
Bezeichner als Schlüsselworte 15ff.
blank 15, 107ff.
bolds 14ff.
bubble sort 52

CAT 25, 108ff.
CDL 11
close 91ff.
CONCR 62ff.
consting 29ff.
compilieren, in line 8
complex 112ff.
compound statements, explizit 6,7
compress 110ff.
condition 40
cos 116ff.

Datei 89ff.
    Assoziierung 90ff.
    Betriebsrichtung 90ff.
    Deklaration 90ff.
    direkte 89ff.
    sequentiell 89ff.
    wahlfreie 89ff.
Datenabstraktion, starke 10, 61ff.
    schwache 48ff.
Datenobjekt 17ff.
    CONST 20
    VAR 20
    Vereinbarung von 20ff.
    zusammengesetztes 8, 31ff.
Datentypen 17ff.
    abstrakte 10, 61
    einfache 6, 17ff.
    neudefinierte 10, 17, 61ff.
    Zusammenfassung von gleichartigen 31ff.
    Zusammenfassung von ungleichartigen 31, 34ff.
DECR 25, 104ff., 105ff.
Deklaration 17, 19, 20ff.
    mit Initialisierung 28ff.
delete 101ff.
Denoter 6, 18ff.
    BOOL 19
    einfacher Datentypen 18ff.
    für abstrakte Datentypen 62ff.
    für Reihungen 32ff.
    für Strukturen 35ff.
    INT 18
    REAL 18
    TEXT 19
Denotierungs-Prozedur 71ff.
DIRFILE 89ff.
DIV 25
Division 25

Einheiten 39ff.
    Aneinanderreihung von 39ff.
    Werte liefernde 55ff.
Elaboration 22ff.
    eines Operators 23ff.
    Reihenfolge der 26ff.
ELAN 4
    Entwurfskriterien 13
    Überblick über die Entwicklung 11ff.
Elemente, lexikalische 14ff.
Entwurfskriterien 13
eof 102ff.
eop 102ff.
erase 92ff.
errorstop 122ff.

exp 117ff.
Exponentation 25
exklusives oder 25
expression language 7

Fachsprache 88ff.
Feinstruktur 61ff.
    eines Strukturtyps 35
    Zugriff auf die 62ff.
Felder von Strukturen 34ff.
FILE 89ff.
FOR-Konstrukt 46ff.
Formel 26ff.
    Auswertung einer 27ff.

Generizitaet 9
    von Operatoren 23ff.
    von Prozeduren 69ff.
get 92ff.
getline 98ff.
Gleichung, quadratische 52
Gültigkeitsbereich von Objekten 80ff.

identifier 15ff.
INCR 25, 104ff., 105ff.
Index 32
Indizierung 32
Initialisierung 6, 17, 28ff.
    dynamische 47ff.
    -Symbol 28
Interface 77ff.

Klammern 27ff.
Komponenten
    einer Reihung 31
    von Strukturen 35ff.
Kommentare 16ff.
    verschachtelte 17ff.
    -Klammern 16ff.
Konkretisierer 62ff.
Konstrollstruktur 7
Konstruktor 9, 35ff., 62ff.

Länge eines TEXTs 24
LEAVE-Konstrukt 59ff.
Leerzeichen 15
LENGTH 24, 108ff.
Leserecht 6
Lese-/Schreibrecht 6
LET-Konstrukt 48ff.
liefern 22
line 100ff.
linenr 102ff.

literal 6
ln 117ff.
logisches oder 25
logisches und 25

main packet 78ff.
max 104ff, 106ff.
maxint 104ff.
maxlinelength 102ff.
maxpagelength 102ff.
maxreal 105ff.
maxtextlength 107ff.
min 104ff., 106ff.
minint 104ff.
MOD 25, 104ff., 105ff., 121ff.
Modul 9, 10, 75ff.
Multiplikation 25

Namen 14, 15ff.
Negation 24
new 102ff.
niltext 107ff.
NOT 24

Objekte 22
oder, exklusives 25
oder, logisches 25
Operand 22
    Datentyp eines 24
Operationen 22
    zusammengesetzte 39ff.
Operator 22ff.
    dyadischer 22ff., 24
    Elaboration 23ff.
    für einfache Datentypen 22ff.
    Generizität von 23ff.
    monadischer 22ff., 24
    neudefinierte 10, 71ff.
    neudefinierter dyadischer 72ff.
    neudefinierter monadischer 71ff.
    Prioritäten von 26ff.
    Priorität von neudefinierten 72ff.
    Repräsentation von 24
    Resultat von 23ff.
    Symbol 22
    Wert von 23ff.
    Wirkung eines 24
    Zuweisungs- 24
OR 25

page 100ff.
Paket 9, 75ff.
    vorübersetzt 10
    main 78ff.
    Reihenfolge von 78ff.
    rumpf 78ff.
    Standard 103ff.
    Struktur von 78ff.
Parameter 9, 67ff.
    aktuelle 67
    CONST - 67
    formale 67
    Prozeduren als 68
    - Spezifikation 67
    VAR - 68
PASCAL 11
pos 112ff.
Primärausdrücke 39
Prioritäten 26ff.
Programmieren
    im Großen 13
    im Kleinen 13
Programmiersprache, anweisungsorientiert 7
Programmstruktur 78ff.
Prozedur 9
    als Parameter 9, 68ff.
    Aufruf 65ff.
    Deklaration 65ff.
    Denotierungs - 71ff.
    Generizität von 69ff.
    geschachtelte 65
    - Parameter 9, 68ff.
    rekursive 65ff.
    -rumpf 66
    Werte liefernde 70ff.
put 96ff.
putline 99ff.

random 117ff.
range 80ff.
    extended - 80ff.
Refinement 7, 50ff.
    Anwendung 50, 54
    Definition 54ff.
    Einsetzung von 54
    Konstruktion 54ff.
    Trennung von 54ff.
    Werte liefernde 57ff.

Reihung 31ff.
   Anzahl der Komponenten einer 31
   Denoter 32ff.
   dynamische 31
   Operationen auf 32ff.
   -styp 17
   von Reihungen 33ff.
   Zugriff auf die Komponenten 32ff.
   Zuweisung von 32
replace 111ff.
reserved symbols 14ff.
reset 101ff.
Resultat 23
   Accessrecht eines 23ff.
   Datentyp eines 24
round 106ff.
ROW 31
row display 9, 32ff.

Schlüsselwörter 14ff.
Schleife 44ff.
   abweisende 44ff.
   endlose 44ff.
   nichtabweisende 45ff.
   Zähl- 46ff.
Schnittstelle 10, 77ff.
Selektion 35ff.
Selektor 35ff.
Sequenz 7, 39ff.
SIGN 24, 104ff., 105ff.
sin 115ff.
SLAN 11
smallreal 105ff.
smallestreal 105ff.
Sonderzeichen 15ff.
sqrt 117ff.
Speicherplatz eines Datenobjekts 17ff.
Standard
   definierter 89ff., 103ff.
   erweiterter 89ff., 103ff.
   Pakete 103ff.
statement language 7
statement separator 7
stop 122ff.
Strukturen 8, 34ff.
   Datentyp von 34ff.
   Denoter für 35ff.
   Felder von 34ff.
   Reihungen innerhalb von 36ff.
   Reihungen von 37ff.
   -styp 17
   von Strukturen 36ff.
   Zugriff auf die Komponenten von 35ff.
   Zuweisung von 35ff.

SUB 25, 111ff.
Subskription 9, 32ff.
subtext 110ff.
Subtraktion 25

tan 117ff.
Terminator 59ff.
    für Prozeduren und Operatoren 74ff.
Text
    Länge von 24
    Prozedur 118ff.
    Teil- 25
    Vervielfältigung von einem 25
top town Methode 13
trunc 105ff.
Typ
    -anpassung 29
    -schutz 8

Überdeckungen 82ff.
und, logisches 25
unit 39ff.
UNTIL-Konstrukt 45ff.

Vereinbarung 17
    von Datenobjekten 20ff.
Vervielfältigung von TEXTen 25
Verfeinerung, schrittweise 50ff.
Vorzeichenwechsel 24
void 7

Wandlung des Accessrechts 29ff.
Wert
    eines Datenobjekts 17ff.
    liefernde Einheiten 55ff.
    undefinierter 17
WHILE-Konstrukt 44ff.
Wiederholung 7, 44ff.

XOR 25

Zeichen
    -folge 14ff.
    -satz 14ff.
Zugriffsalgorithmen 10
Zuweisung 27ff.
    von Reihungen 32ff.
    von Strukturen 35ff.
    -soperator 24, 25